NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

DYNAMICALLY EXTENDING A NETWORKED VIRTUAL ENVIRONMENT USING BAMBOO AND THE HIGH LEVEL ARCHITECTURE

by

Stewart W. Liles

September 1998

Thesis Advisor:	Michael Zyda
Associate Advisor:	Rudy Darken

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE September 1998	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
Dynamically Extending a Networked Virtual Environment Using Bamboo and the High Level Architecture

5. FUNDING NUMBERS

6. AUTHOR(S)
Liles, Stewart W.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The design and execution of a networked virtual environment (NVE) are challenging tasks made even more difficult by the fact that NVEs are becoming more complex and difficult to manage. In a distributed environment, each simulation not only computes its own behaviors and publishes them to the network, but it must accurately represent all other entities participating in the NVE. To simplify this task, this thesis implements method to make distributed simulations dynamically extensible, flexible, specific, and consistent. Bamboo provides the ability to dynamically extend the virtual environment by defining a convention by which plug in modules can be added during simulation runtime. The HLA provides the network communication layer that transports entity state updates to all members of the distributed simulation. These two tools combine to create a unique solution to problems inherent in designing modern networked virtual environments. The implementation is dynamically extensible which increases the flexibility implementers have in designing virtual environments. The HLA transports the entity updates and the module name that must be used to represent the entity. This method allows programmers to design only their module because modules representing other entities will load as needed during the execution. This method of implementing virtual environments that promises to streamline the design and implementation process.

14. SUBJECT TERMS
Network Virtual Environment, HLA, Bamboo

15. NUMBER OF PAGES
93

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

DYNAMICALLY EXTENDING A NETWORKED VIRTUAL ENVIRONMENT USING BAMBOO AND THE HIGH LEVEL ARCHITECTURE
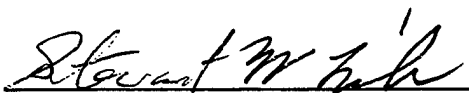
Stewart W. Liles
Captain, United States Army
B.S., Oklahoma State University, 1988

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE MODELING VIRTUAL ENVIRONMENTS AND SIMULATION

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

Author: _____

Stewart W. Liles

Approved by: _____

Michael Zyda, Thesis Advisor

_____

Rudy Darken, Associate advisor

_____

Michael Zyda, Academic Associate
Modeling Virtual Environments and Simulation
Academic Group

iii

ABSTRACT

The design and execution of a networked virtual environment (NVE) are challenging tasks made even more difficult by the fact that NVEs are becoming more complex and difficult to manage. In a distributed environment, each simulation not only computes its own behaviors and publishes them to the network, but it must accurately represent all other entities participating in the NVE. To simplify this task, this thesis implements method to make distributed simulations dynamically extensible, flexible, specific, and consistent. Bamboo provides the ability to dynamically extend the virtual environment by defining a convention by which plug in modules can be added during simulation runtime. The HLA provides the network communication layer that transports entity state updates to all members of the distributed simulation. These two tools combine to create a unique solution to problems inherent in designing modern networked virtual environments. The implementation is dynamically extensible which increases the flexibility implementers have in designing virtual environments. The HLA transports the entity updates and the module name that must be used to represent the entity. This method allows programmers to design only their module because modules representing other entities will load as needed during the execution. This method of implementing virtual environments that promises to streamline the design and implementation process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

A. MOTIVATION

The design and execution of a networked virtual environment (NVE) are challenging tasks made even more difficult by the fact that NVEs are becoming more complex and difficult to manage. In a distributed environment, each simulation not only computes its own behaviors and publishes them to the network, but it must accurately represent all other entities participating in the NVE. To simplify this task, a method must be devised to make distributed simulations dynamically extensible, flexible, specific, and consistent. A dynamically extensible virtual environment would allow users to change the executable at runtime to whatever state is specified by the user. The challenge is to design a system that allows users to design entity definitions that can be loaded and unloaded during execution. With true dynamic extensibility comes flexibility, and designers are not tied to compile time determinations of behavior. Consistency in this sense means all sites participating in the NVE need the same definitions for each entity and the terrain model being used. Finally, the designer of a specific simulation, such as a tank simulator, should not need to design the other entities that will be depicted in the simulation. These remote objects should be program objects that can be added as needed during execution. This approach will allow NVE implementers and programmers the flexibility to design and run NVEs in real time without the problems of inflexibility and static design inherent to distributed simulation.

B. BACKGROUND

There are many examples of NVEs that use different methods of communicating entity state and ensuring consistency between simulation locations. Examples include DIVE [1] and BRICKNET [2]. These systems share one characteristic. They are defined at compile time and are unchangeable during execution. They allow dynamic allocation of memory but the definitions of each entity are defined at compile time and are unchangeable.

The DIVE core uses peer-to-peer communication between shared virtual worlds. All DIVE processes connected to the same world are identical. A DIVE process can choose what world it is a part of but it can only be a member of one world at a time. DIVE is limited by the paradigm of distributed, shared memory. This paradigm creates significant network traffic trying to keep the shared memory consistent from process to

1

process. While DIVE shared virtual worlds may change during runtime, the method of communication and the abilities of the system are previously defined and are not extensible at runtime.

BRICKNET allows the exchange of objects and object updates through BRICKNET servers. While it does distribute processing on multiple servers, entities require the server to update state and define behavior. Workstations are primarily used to render the graphics for the user. Furthermore, BRICKNET object updates are predefined and the update protocol cannot be modified.

Finally, distributed interactive systems (DIS) like NPSNET [3] are designed on the premise that each site could build their own simulation and choose how to represent each type of entity without regard to the consistency of this representation across the network. Each site could have its own terrain model and its own representations for each entity type. Because of these inconsistencies, DIS simulations are plagued with discrepancies between entity position and orientation and line of site computations related to the terrain models. DIS is highly enumerated and the packets containing entity state are large and mostly redundant. These inconsistencies complicate interactions between entities because the terrain may provide different line of sight computations from one simulation to the next. Additionally, the actual polygonal representation and behaviors of the entity may not be consistent among workstations in the distributed simulation. This causes excess network traffic to solve simple interactions between entities and keep entity state updated accurately between simulation sites.

Until now, there was no way to ensure all simulation sites had the proper polygonal and behavioral representation for all entities participating in the virtual environment. A new system, Bamboo [4], provides such a capability by providing simulations a framework to dynamically load and unload program modules as the situation changes in the virtual environment. High Level Architecture (HLA) [5] replaces DIS and provides the network communication capabilities. By implementing the simulation as a group of program modules, designers solve the problem of ensuring that every site running in the NVE is consistent with every other. The designer just ensures every participant in the NVE knows the network location of all the program modules making up the NVE. Then, as the virtual world executes, each site loads and unloads modules as needed. All simulation sites have the same representations for each entity as well as its associated behaviors and controls.

This thesis describes an implementation that uses Bamboo to handle the dynamic nature of modern NVEs, and HLA to handle the communication between each simulation site. The following sections provide an overview of Bamboo and HLA features and how

2

they apply to this implementation. Later chapters provide a detailed description of both systems.

C.   BAMBOO

Bamboo enables dynamically scaleable virtual environments hosted on a network. It achieves this goal by an efficient implementation that provides direct support for the key issues pertaining to VE development. These issues include dynamic extensibility, module dependency, and event handling [4]. Bamboo's most notable feature is its ability to dynamically extend its capabilities during run time. It achieves this goal by implementing a plug-in metaphor similar to that used by commercial packages like PhotoShop [6] and Netscape [7]. In addition to the plug-in metaphor, Bamboo implements a system that allows the user to extend the executable through a series of callbacks that a newly added module allocates at load time. The event handler simply provides an abstraction for handling system-generated events. The event handler uses the callback handler to notify registered parties of an event. Bamboo receives this notification as a callback. Module dependency provides a system to ensure that modules which are required by other modules are loaded first before the depending module. Bamboo uses callback handlers so multiple callbacks respond to a single event. This is a cursory introduction to the features and capabilities of Bamboo. Chapter two provides a detailed description of the Bamboo system.

D.   HIGH LEVEL ARCHITECTURE

The High Level Architecture (HLA) is the Department of Defense standard for simulation interoperability [5]. HLA is not software. It is an architecture that provides standard methods of defining how distributed simulations will communicate. It is a set of specifications that define data objects. These standards are specified in the HLA Interface Specification and the Object Model Template (OMT). The HLA Interface Specification defines the interface between the simulation and the software that will provide the network and simulation management services. The Runtime Infrastructure (RTI) is the software that provides these services. The OMT prescribes a common method for recording the information that will be produced and communicated by each simulation participating in the distributed exercise. This discussion of HLA is continued in detail in Chapter 3.

E.   THESIS ORGANIZATION

This thesis is organized into the following chapters:

- Chapter I: Introduction. Outlines the organization of the thesis and addresses the significance of introducing and evaluating a new method for implementing a networked virtual environment.

- Chapter II: Bamboo. Discusses in detail the current implementation of Bamboo and how its capabilities are suited for a networked virtual environment implementation.

- Chapter III: High Level Architecture. Discusses in detail the current implementation of the HLA and how its capabilities are suited for a networked virtual environment implementation. This chapter includes a detailed discussion of the run time infrastructure (RTI) and how its capabilities are exploited in this implementation.

- Chapter IV: Implementation. Describes the development of the experimental virtual environment that illustrates the power of combining Bamboo and HLA.

- Chapter V: Results and Limitations. Describes performance results for the implementation and certain limitations discovered during development.

- Chapter VI: Conclusions and Recommendations. Discusses the significance of the results and gives ideas as to future work that should be completed in this area.

II.   BAMBOO

A.   INTRODUCTION

Bamboo enables dynamically scalable virtual environments hosted on a network [4]. It achieves this goal by an efficient implementation that provides direct support for the key issues pertaining to VE development. These issues include dynamic extensibility, event handling, and module dependency. By addressing these issues, Bamboo provides the ability for the system to dynamically configure itself during runtime. Specifically, this framework provides the ability to discover simulation objects on the network at runtime and automatically load the correct module to represent the entity.

B.   DYNAMIC EXTENSIBILITY

Bamboo's most notable feature is its ability to dynamically extend its capabilities during run time. It achieves this goal by implementing a plug-in metaphor, then extends the idea by adding module dependency, a generalized method of extending the executable and a generalized event handler.

1.   Dependency

Bamboo extends the plug-in metaphor by adding inter-module dependencies. Tracking inter-module dependencies could be complex. Fortunately, as Bamboo loads each module, it verifies that dependent modules load first. If they are not loaded, it automatically loads them without specific interaction with the user. Using Figure 1 as an example, assume M3 is already loaded. If M4 loads later, the system verifies the presence of M2 in memory. Bamboo loads M2 if it is not in memory. As M2 is being loaded Bamboo verifies the presence of M1. M4 finally loads because Bamboo verified all its dependencies [4].

5
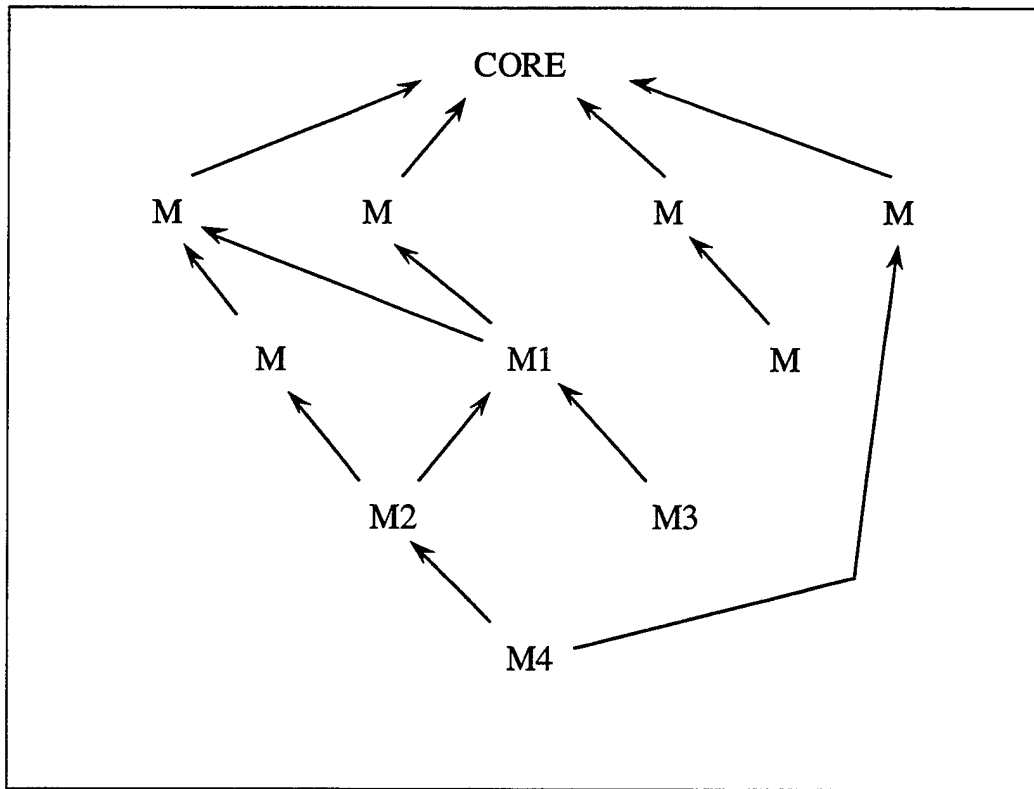
Figure 1: Module Dependency View

2. Extending the Executable

Dynamic loading of program modules does not in itself ensure dynamic extensibility. Bamboo uses a callback handler that allows each module to attach and remove itself from the process's execution loop when being paged in and out of memory. The callback handler derives from objects that can be named so it is easily located and manipulated. The callback itself is recursive and provides two callback handlers, one just before callback execution and one directly after. This allows grouping of like functionality. For example, rendering engines implement some form of app, cull and draw as a pipeline. Users refer to surrounding areas as pre and post app, pre and post cull, and pre and post draw. The executable begins to resemble a tree of callbacks. Figure 2 illustrates how using callbacks and callback handlers to extend the executable begins to resemble a tree of callbacks. For instance, a user module may load itself and depend to the visual and keyboard modules. At load time, the user module defines callbacks that provide execution time for the logic in the module. Furthermore, any pruning or pausing of sub-trees automatically includes its children. Therefore if a callback handler is deleted, all of its associated callbacks are also deleted without specific user interaction.
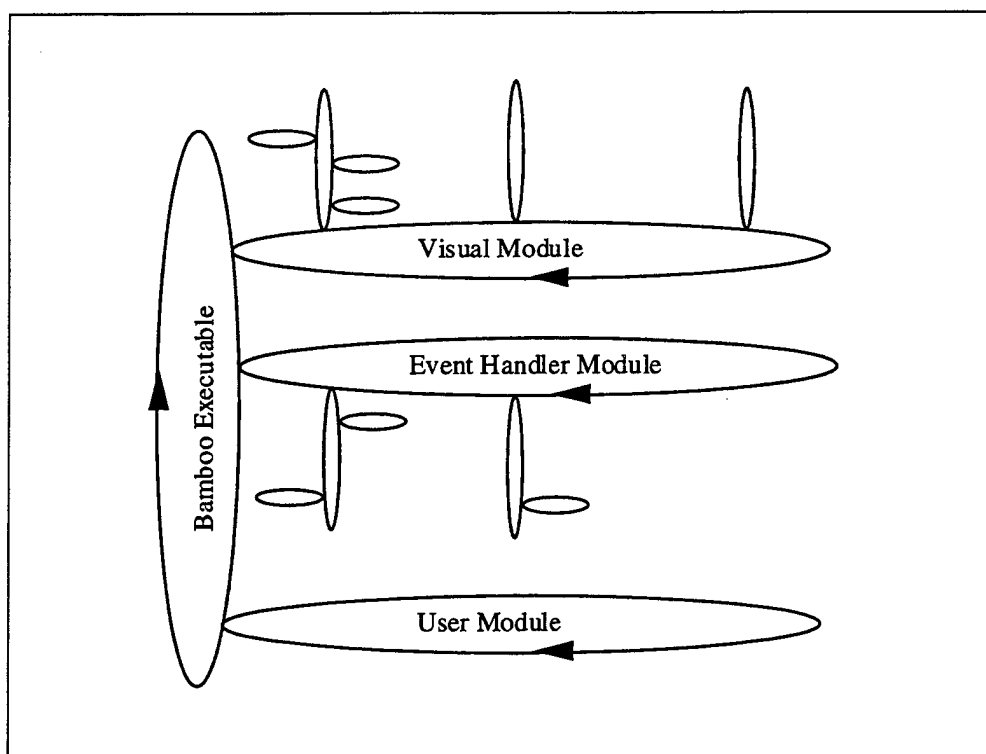
Figure 2: Extending the Executable

3. Event Handling

The event handler simply provides an abstraction for handling system-generated events. The event handler uses the callback handler to notify registered parties of an event. Bamboo receives this notification as a callback. Bamboo uses callback handlers so multiple callbacks respond to a single event. For example, a module that captures keyboard events would monitor the keyboard in a separate thread listening for keys identified by the user. In figure 2, the Event Handler Module has multiple callbacks on two separate keys. When a key is pressed, then the callbacks are executed in the order specified.

C. BAMBOO CONCLUSION

Bamboo improves on the plug-in metaphor in three distinct ways. It provides a convention for the definition of program modules. Second, it generalizes a method to extend the executable, and third, it provides a method to build dependencies between modules. Using these features, Bamboo overcomes many of the pitfalls of monolithic virtual environment architectures by providing modular components and a dynamically extensible runtime executable.

7

III. HIGH LEVEL ARCHITECTURE

A. INTRODUCTION

The High Level Architecture (HLA) provides a common architecture for reuse and interoperation of simulations. This means that simulations designed for a specific purpose may be reused in a different application using the HLA concept of a federation: a composable set of interacting simulation participants - federates. The intent is to provide a standard architecture under which simulations are designed so that they can be reused thereby reducing the time and cost required creating a new environment for a new purpose. [5]

The Object Model template (OMT) [8], HLA rules [9] and the Interface Specification (IF Spec) [10] define the HLA standard. A final component of the HLA is the Runtime Infrastructure (RTI). The OMT describes the essential sharable elements of the simulation or federation in 'object' terms. In the HLA sense, objects are collections of attributes that describe simulation entity state that are communicated between federates operating in the federation. Second, the IF Spec describes the runtime services provided to each federate by the RTI. The RTI is the software component of the HLA that supports the exchange of data defined by the OMT component. Finally, The HLA rules summarize the key principles behind the HLA.

B. OBJECT MODEL TEMPLATE

The HLA is designed to facilitate interoperability. Hence, the OMT is designed to provide a means for open information sharing across the simulation community. The OMT does not constrain the content, but provides a streamlined format for communicating to the other users, who may reuse the simulation, and the data inputs and outputs of the simulation. The HLA specifies two types of object models: the HLA Federation Object Model (FOM) and the HLA Simulation Object Model (SOM). The FOM is a specification of the exchange of public data among the participants in a HLA federation. Those participants are called federates. The HLA FOM describes the set of objects, their attributes and interactions that are shared between federates in a federation. The SOM describes the simulation (federate) in terms of the types of objects, attributes, and interactions it can offer to future federations.

1. Federation Object Model

The FOM requires information concerning the object classes, object interactions, class attributes, interaction parameters and a lexicon describing each of the above [8].

a. Object Class Structure Table

Each object class must be named and its relationship to other classes must be defined. All classes must be defined and described in the lexicon. The result is a table similar to Table 1 [8].

Object Class Structure Table			
<class>(<ps>)	[<class> (<ps>)]	[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
		[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
		...	...
		[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
	[<class> (<ps>)]	[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
		...	...
		[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
	...	...	
<class>(<ps>)	[<class> (<ps>)]	[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
		[<class> (<ps>)]	[<class> (<ps>)] [,<class> (<ps>)]*l [<ref>]
		...	...
...	...	...	...
Air Vehicle(S)	Fixed Wing (S)	Fighter-Attack (S)	F-14 (PS)
			F-16 (PS)
			F-18 (PS)
		Bomber (S)	B-1B (PS)
			B-2 (PS)
	Rotary Wing (PS)		

Table 1: Object Class Structure Table

This table shows the object model used in a federation. Class Air Vehicle is the base class for all flying entities. Class Fixed Wing and Rotary wing inherit from Air Vehicle, while Fighter-Attack and Bomber inherit from Fixed Wing. Then F-14, F-16, etc. describe the specific types of aircraft depicted in the federation. The (P), (S) and (PS) stand for publishable, subscribable and both, respectively. This means that individual federates can request – subscribe - certain object attributes from the federation. A federate may also provide – publish - object attributes to the federation. If a federate represents an entity that must interact with entities of the same type, that federate might publish and subscribe to the same Object Class.

### b.	Object Interaction Table

The object interaction table shows the interaction class structure for the federation. Table 2 shows the interaction class structure [8]. An interaction is an explicit action taken by an object that can optionally be directed toward other objects. In this case, Weapon Detonate is the base interaction class, and Weapon Detonate at Air Target and Weapon Detonate at Ground Target are both inherited from Weapon Detonate. This table also lists the initiating and receiving object and the affected attributes for each object.

Object Interaction Table							
Interaction Structure		Initiating Object		Receiving Object/Area		Interaction Parameters	Init/ Sense/ React
		Class	Affected Attributes	Class	Affected Attributes		
<interaction>	[<interaction>]	<class> [,<class>]*	[<attribute>] [,<attribute>]* [(<comment>)]*	[<class>] [,<class>]*	[<attribute>] [,<attribute>]* [(<comment>)]*	[<parameter>] [,<parameter>]*	<isr>
	[<interaction>]	<class> [,<class>]*	[<attribute>] [,<attribute>]* [(<comment>)]*	[<class>] [,<class>]*	[<attribute>] [,<attribute>]* [(<comment>)]*	[<parameter>] [,<parameter>]*	<isr>
--	--	--	--	--	--	--	--
<interaction>	[<interaction>]	<class> [,<class>]*	[<attribute>] [,<attribute>]* [(<comment>)]*	[<class>] [,<class>]*	[<attribute>] [,<attribute>]* [(<comment>)]*	[<parameter>] [,<parameter>]*	<isr>
--	--	--	--	--	--	--	--
Weapon Detonate	Weapon Detonate at Air Target	Weapon	Velocity, Acceleration, Weight, :	Air Vehicle	Velocity, Acceleration, Weight, :	Weapon Location, Warhead, Weapon Attitude,	IR
	Weapon Detonate at Ground Target	---	---	---	---	---	---

Table 2: Object Interaction Table

The last column in the table shows the three basic categories of interaction:

- Initiates (I): indicates that a federate is currently capable of initiating and sending interactions of the type specified in that row.

- Senses (S): indicates that a federate is currently capable of subscribing to the interaction and utilizing the interaction information, without necessarily being able to effect the appropriate changes to affected objects.

- Reacts (R): indicates that federate is currently capable of subscribing and properly reacting to the interactions of the type specified by effecting the appropriate changes to any owned attributes of the affected objects.

In a FOM definition, all of the above are valid entries. There would not be a listing if there was not a federate responsible for the interaction.

11

a. Attribute/Parameter Table

Finally, the Attribute/Parameter Table [8] defines characteristics pertinent to each Class/Interaction described in Table 1 and Table 2. Interactions are published by and subscribed to by federates depending on the structure of the federation. Table 3 is the Attribute/Parameter Table for a Tank Object and the Weapon Detonate Interaction. For each Object/Interaction the

Object/ Interaction	Attribute/ Parameter	Data Type	Cardinality	Units	Resolution	Accuracy	Accuracy Condition	Update Type	Update Condition	T/A	U/R
<class> I <Interaction>	<attribute>I <parameter>	<datatype>	[<size>]	<units>	<resolution>	<accuracy>	<condition>	<type>	<rate> I <condition>	<ta>	<ur>
	<attribute>I <parameter>	<datatype>	[<size>]	<units>	<resolution>	<accuracy>	<condition>	<type>	<rate> I <condition>	<ta>	<ur>
	...	...	[<size>]	...	...	...	...	...	...	...	...
<class> I <Interaction>	<attribute>I <parameter>	<datatype>	[<size>]	<units>	<resolution>	<accuracy>	<condition>	<type>	<rate> I <condition>	<ta>	<ur>
	<attribute>I <parameter>	<datatype>	[<size>]	<units>	<resolution>	<accuracy>	<condition>	<type>	<rate> I <condition>	<ta>	<ur>
	...	...	[<size>]	...	...	...	...	...	...	...	...
<class> I <Interaction>	<attribute>I <parameter>	<datatype>	[<size>]	<units>	<resolution>	<accuracy>	<condition>	<type>	<rate> I <condition>	<ta>	<ur>
...	...	...	[<size>]	...	...	...	...	...	...	...	...
Tank	Area	Float	1	m2	0.1 m2	perfect	always	conditional	scen events	TA	UR
	Velocity	Double	1	m/sec	1 m/sec	.01 m/sec	none	periodic	10 Hz	TA	UR
	State	Tank_Type	1	n/a	n/a	n/a	n/a	conditional	scen events	TA	UR
	Position	Rectng_Type	1	n/a	n/a	n/a	n/a	periodic	10 Hz	TA	UR
Weapon Detonate	Warhead	Wh_Type	1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Table 3: Attribute/Parameter Table

attributes or parameters are defined and characterized by multiple descriptors like data type and units of measure. The second to last column, Transferable/Acceptable refers to a federate's ability to transfer or accept the responsibility to update the specified attributes for a particular entity. For a FOM, the only acceptable entries are (TA) for Transferable/Acceptable or (N) for Not Transferable/Acceptable. The last column refers to a federate's capability to update (U) and reflect (R) attributes or parameters. In a FOM, all attributes or parameters should be marked both updatable and reflectable.

12

2. Simulation Object Model

A FOM is the union of all SOMs used to define the federation. The SOM uses the same templates as the FOM. The Object Class Table in Table 1 describes the object classes represented in the federation. A SOM object class structure table would designate which classes the federate publishes (P) and which it has the capability to subscribe (S). For example, the F-16 federate might publish the F-16 object and subscribe to air vehicle so it would know the location and speed of all other aircraft in the federation. Similarly, the Object Interaction table would differ in the last column because the federate must identify what interactions it has the capability to process. The F-16 federate might put a (R) in the last column of Table 2 to indicate that it reacts to a Weapon Detonation at an Air Target. Additionally, a (I) might go in the last column for Weapon Detonate at Ground Target to show that the F-16 federate initiates the interaction to fire weapons at a ground target. Finally, the last two columns in the Attribute/Parameter Table would be modified to show what capabilities the federate has in regards to its ability to transfer and accept attributes or update and reflect attributes.

3. FOM/SOM Lexicon

To achieve interoperability between simulations, all data required by the federation must be fully explained. It is not enough to merely specify the classes of data required by the templates above. The semantics of this data must also be explained. The FOM/SOM Lexicon provides a means for federations to document the definitions of all terms utilized during construction of FOMs, and for individual federates to document the definitions of all terms provided in their SOMs.

C. INTERFACE SPECIFICATION

The IF Spec [9] describes the runtime services provided to the federates by the RTI, and from the federates to the RTI. There are six classes of service. Each defines a specific set of functions that pertain to a particular type of transaction that must be accomplished to properly manage the federation. There are two locations where the function definitions reside: the RTI ambassador and the federate ambassador. The RTI ambassador is the software component provided by HLA and contains all the functionality required to accomplish communication to the network. The federate ambassador is the RTI's interface to the federate. To pass data to the federate, the RTI ambassador makes function calls to a user defined federate ambassador. The RTI ambassador is the same for all federates but the federate ambassador is different for each federate. A full explanation

of each function can be found in the HLA IF Spec. The following discussion deals mainly with the purpose of each management service.

1.   Federation Management

Federation management services offer the basic functions required to initiate the federation, add federates and delete federates as the federation finishes execution. These services include Create, Join, Pause/Resume, Resign and Destroy federation.

2.   Declaration Management

Declaration management defines the services required to support efficient management of data exchange. It does this by providing the services that allow federate's to identify to the RTI ambassador the object and interaction classes they will publish and subscribe. This service includes Publish, Subscribe, and Control actions on specific object classes and interactions.

3.   Object Management

Object management services refer to all the functions required to manage the update of object attributes during federation execution. The services include Request Object ID numbers, Update object attributes, Sending Interactions, Receiving object updates, Receiving interactions, Deleting objects and Changing transportation characteristics.

4.   Ownership Management

Ownership management refers to the dynamic transfer of ownership of object attributes during federation execution. The services include Request attribute ownership, Divest Attribute ownership and Release attribute ownership.

5.   Time Management

Time management services support the synchronization of runtime simulation data exchange. The current HLA time management service provides support for time-step and event-driven simulation systems but support for platform level real-time simulations is limited to wall clock time.

6. Data Distribution Management

Data distribution management supports the efficient routing of data among federates during the course of a federation execution. This service allows federates to identify regions of interest and limit attribute to entities that fall into those regions.

D. HLA RULES

There are ten basic rules that define the responsibilities and relationships between HLA federates and the federation. There are ten rules: five rules apply to federations and five rules apply to federates [10].

1. Federation Rules

- Rule 1: Federations shall have an HLA FOM, documented in accordance with the HLA OMT.

- Rule 2: In a federation, all object representation shall be in the federates, not in the runtime infrastructure. This means that the RTI cannot be used to track entity state. All entity representations are defined by the federate and communicated to other federates via the RTI.

- Rule 3: During a federation execution, all exchanges of FOM data among federates shall occur via the RTI.

- Rule 4: During a federation execution, federates shall interact with the RTI in accordance with the HLA IF Spec. The only way to interface with the RTI is through the RTI ambassador and the services provided in that class

- Rule 5: During a federation execution, an attribute of an instance of an object shall be owned by only one federate at any given time. No attribute can be published by more than one federate at a time.

2. Federate Rules

- Rule 6: Federates shall have an HLA SOM documented in accordance with the HLA OMT. Each simulation must describe the functionality it will provide to the federation. The federation is not required to use all the functionality supplied by the federate.

- Rule 7: Federates shall be able to update and/or reflect any attributes of objects in their SOM and send and/or receive SOM object interactions externally, as specified in their SOM.

- Rule 8: Federates shall be able to transfer and/or accept ownership of attributes dynamically during a federation execution, as specified in their SOM.

15

- Rule 9: Federates shall be able to vary the conditions (e.g., thresholds) under which they provide updates of attributes of objects, as specified in their SOM.

- Rule 10: Federates shall be able to manage local time in a way that will allow them to coordinate data exchange with other members of a federation. Non-time managed federates manage time internally in their own way, but time managed federates must manage time in such a way so it appears there is only one clock.

E. HLA CONCLUSION

The HLA architecture is designed to improve the method of simulation interaction by making certain types of simulation systems reusable among disparate simulation applications. It accomplishes this goal by providing a standard method of communicating simulation capabilities (SOM) and a standard method of defining how those simulations will communicate on the network (FOM). The RTI provides the communications link to manage the operation of a federation using the IF Spec. The HLA rules link it all together by providing guidelines that ensure designers will use all the components is such a way as to accomplish the goal of reusable inter-operating simulations

IV. IMPLEMENTATION

A. INTRODUCTION

The goal is to provide a dynamic, flexible, and consistent environment for three-dimensional networked virtual environments. Consistency is accomplished by ensuring all modules are available locally or via the network from a centralized location. As the simulation executes, users decide which terrain module to use and which simulation entities will populate the environment. Bamboo provides the ability to dynamically load and unload modules. The ability to add a module provides dynamic extensibility. It is the ability to unload modules that makes the system flexible. Flexibility requires that the implementers can change the virtual environment on the fly without restarting.

To this end, the implementation has three major components: the HLA administration module (amHLAAdmin), entity modules, and terrain modules. The amHLAAdmin module manages the communication layer (RTI), module loading and unloading, and all participating entity objects. Each entity module represents the behavior and polygonal representation for each entity. The terrain modules are the same as the entity modules but they must be identified as terrain for the amHLAAdmin module. These modules make up the core of this implementation. Each component is described in detail in the following sections.

1. HLA Administration Module (amHLAAdmin)

The amHLAAdmin module manages HLA RTI communications, Bamboo function calls, the execution window, and entities in the environment. All modules designed for the implementation depend on the amHLAAdmin module. Therefore, Bamboo ensures it loads before any entity or terrain module. Similarly, the amHLAAdmin module depends on various modules being loaded before it. Figure 3 shows the module dependency tree for a typical execution. Notice that amEntity and amTerrain depend on amHLAAdmin, and that amHLAAdmin depends on Visual, Keyboard, and amPageModule.
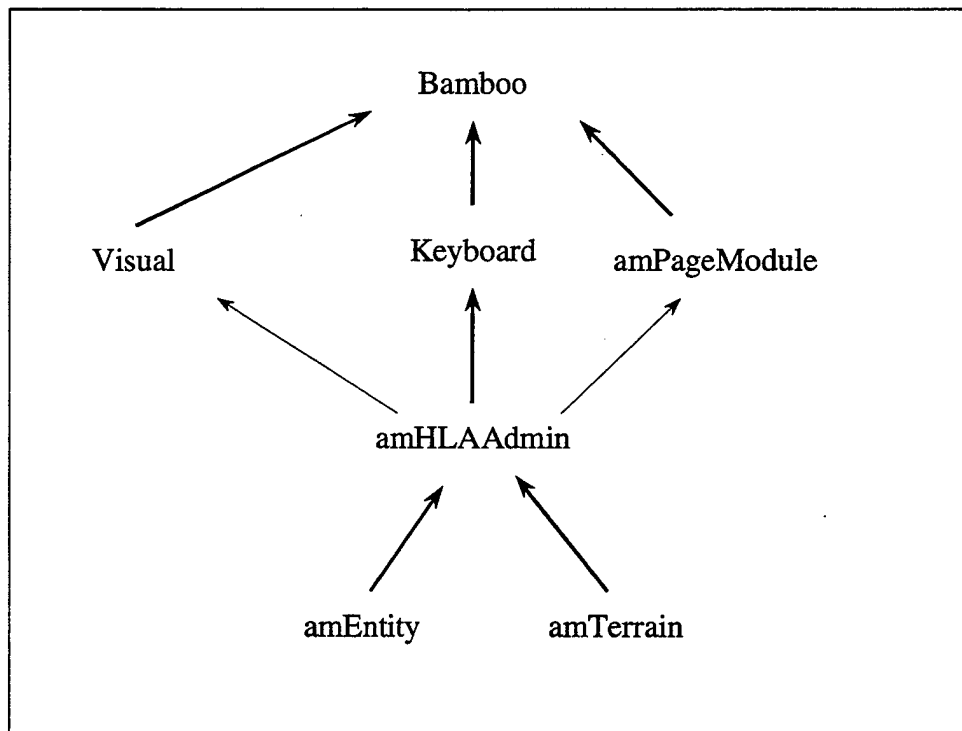
Figure 3: Implementation Module Dependency View

Since the amHLAAdmin module must communicate with the RTI and entity/terrain modules, the API and pure virtual class provide the interface to accomplish this communication. Essentially, the amHLAAdmin module instantiates the Admin class object. Through static functions, the Admin class provides the API for entity/terrain modules to communicate with the RTI and manage instances of their entities. Each entity/terrain class must inherit from amObject. This object defines the interface for the Admin object to communicate with entity/terrain modules.

Figure 4 shows the object model used by this implementation. This figure illustrates the relationship between the Admin object and the entity modules. It also shows how the pure virtual class amObject is used to ensure that each federate transmits and receives all information pertinent to the federation. An entity update coming from the network begins in the RTI ambassador receive buffer.
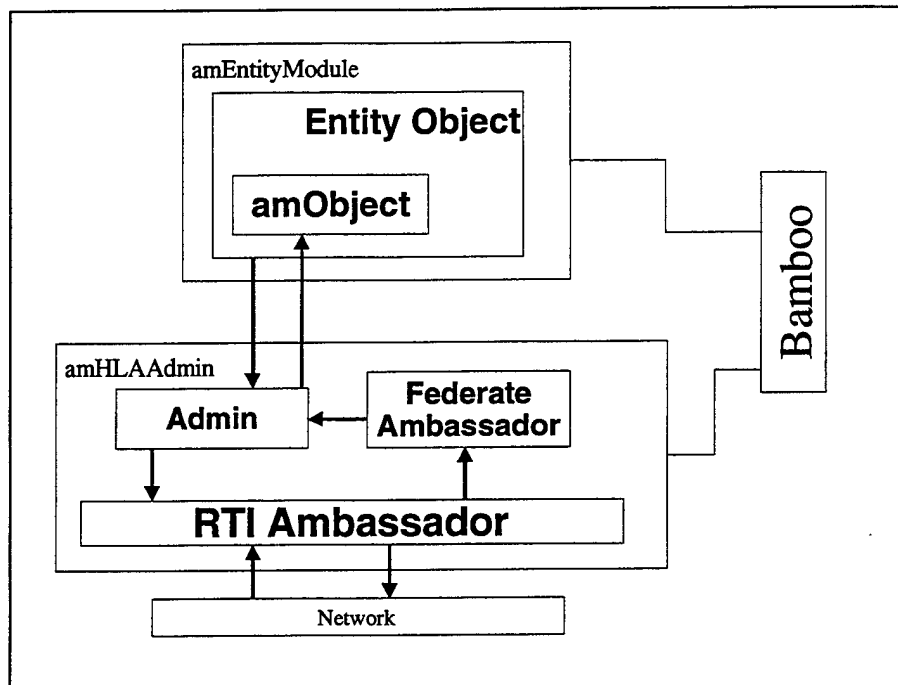
Figure 4: Object Interface

Next, the RTI ambassador calls the AdminFederateAmbassador-::reflectAttributeValues() function in the federate ambassador). The federate ambassador is the RTI ambassador's interface into the federate. This function simply calls the Admin object function to pass the AttributeHandleValuePair to the appropriate entity.

Called by the RTI Ambassador to pass
the `AttributeHandleValuePairSet` to a specific entity

```
void AdminFederateAmbassador::reflectAttributeValues
        ( RTI::ObjectID                          theObject,
          const RTI::AttributeHandleValuePairSet& theAttributes,
          RTI::FederationTime                     theTime,
          const RTI::UserSuppliedTag              theTag,
          RTI::EventRetractionHandle              theHandle )
        throw (RTI::ObjectNotKnown,
               RTI::AttributeNotKnown,
               RTI::InvalidFederationTime,
               RTI::FederateInternalError)
{
    Admin::receiveUpdate(theObject,theAttributes,theTime,theTag,theHandle);
}// end reflectAttributeValues
```

`ObjectID` is the ID number for the specific entity
The `UserSuppliedTag` specifies the module name.
The other parameters are not used in this implementation.

Figure 5: Federate Ambassador Code Fragment

19

The federate ambassador calls the Admin::receiveUpdate() function in the Admin object (Figure 6). This function locates the entity object, an amObject type, in the object list. If the object exists, it is updated. If it does not exist, the Admin object checks to see if the module is loaded. If the module is loaded, then another object representing that entity is instantiated. If the module is not loaded, it is loaded and an object representing the entity is instantiated.

Called by the Federate Ambassador to pass the `AttributeHandleValuePairSet` to a specific entity managed by the Admin object

```
void Admin::receiveUpdate( RTI::ObjectID              theObject,
              const RTI::AttributeHandleValuePairSet&  theAttributes,
              RTI::FederationTime                      theTime,
              const RTI::UserSuppliedTag               theTag,
              RTI::EventRetractionHandle               theHandle )
{
```

Locates amObject in list

```
amObject* tmp = Admin::findSimEntity(theObject);      // find the correct object

if (tmp)
```
Pass attributes to amObject
```
    tmp->receiveUpdates(theObject,theAttributes,theTime,theTag);
else {        // add module if necessary or add simEntity
    if (Admin::moduleLoaded(theTag)){     // is module loaded
```
Add entity of module already loaded
```
        cout << "adding new " << theTag << endl;
        Admin::addSimEntity(theTag,theObject);
    }// end if
    else {
```
Load module and add entity from module
```
        cout << "Loading Module " << theTag << endl;
        Admin::loadModule(theTag);
        Admin::addSimEntity(theTag, theObject);
} } }
```

Figure 6: Admin Object Code Fragment

The Admin object iterates the list of entities and calls the EntityObject::receiveUpdates() function of the appropriate entity (Figure 7). This function is defined in the amObject pure virtual class. This function iterates the AttributeHandleValuePair and sets the appropriate state variables in the entity object.

Figure 7: Entity Object Code Fragment

The Admin object is the portion of the amHLAAdmin module that implements the Admin API. This API is fully described in Appendix A: Implementation Tutorial. Users apply the Admin API to ensure their modules are managed as part of the HLA federation and the entities they represent are properly registered and updated by the RTI. Proper use of the API insures that the federation will comply with HLA Rules.

The amHLAAdmin module loads and unloads modules in two ways: either automatically at the request of the system or explicitly at the request of the user. The Admin class defines certain API functions that load and unload modules at the request of a module or the system. Finally, the amHLAAdmin module loads user requested modules using a separate module called the amPageModule on which it depends. This module loads automatically when the amHLAAdmin module loads. The amPageModule executes the Bamboo calls that load and unload user requested modules. It installs two keyboard events that the implementer uses to arbitrarily load and unload modules.

The amHLAAdmin module manages all of the simulation entities. Functionality related to this task are the HLA object management tasks like registering and deleting objects and ensuring state updates transmit to the correct entity. The amObject class, that all objects inherit from, allows the amHLAAdmin module to iterate its list of simulation

21

entities and update each object based on its ObjectID, an identification number provided by the RTI.

Each module's capability means nothing unless the executable is extended to include the new module. Figure 8 illustrates the execution tree used in this implementation. The amHLAAdmin module created the symbols in bold outline when the module loaded. A1 is a callback attached to the main callback handler created by the Bamboo core. A1 "ticks" the RTI to provide CPU time to the RTI ambassador and the federate ambassador. This callback drives the federate by processing all updates and providing them to the correct simulation entity. A2 is a callback attached to the draw callback handler of the Visual module. This callback calls the display function of all simulation objects using a call to a virtual function defined in amObject that all simulation entities must implement. Finally AK is the callback representing all keyboard events that are processed by the amHLAAdmin module.
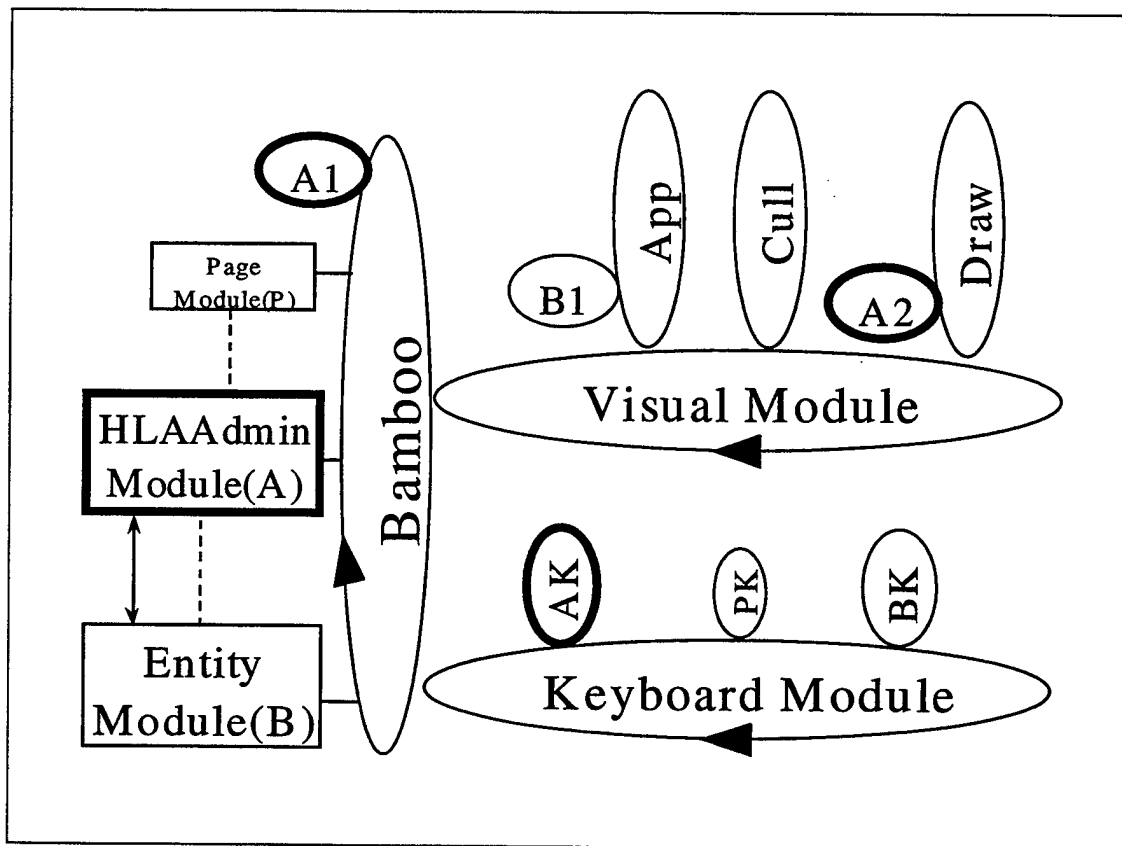
Figure 8: Execution Callback Tree

22

2. Simulation Entities (amEntity/amTerrain)

As the VE executes, if an entity is updated that is not currently represented on the local machine, the RTI initiates the discovery process. The UserSuppliedTag, a character string that is transmitted with each update handle value pair, represents the module name. The handle value pair is the HLA method of creating a byte array for transmission on the network. If this module is already loaded, then another object from this module is instantiated. If the module does not exist, then Bamboo loads it and instantiates an object that represents the newly discovered entity.

Each simulation entity is a Bamboo module. Each module has two major components: the object's polygonal representation and its general behavior. Therefore, when a module loads as a result of a remote object update, the user collects all the controls of that module. Then Bamboo plugs the module into the local event loop so local processing can compute entity appearance and behavior. Figure 4 shows the entity module loaded and inserted in the executable with two sets of callbacks. B1 is the preapp callback that gives the user the ability to control the object with keyboard input. BK is the callback for all the keyboard inputs defined by the module.

Because this system passes behaviors along with polygonal representation, there is an opportunity to reduce network traffic by reducing the details of entity behavior that previous systems transmitted via the network. This occurs because each entity computes its behavior locally not from a remote location. For instance, each entity provides collision event behavior locally without the need for multiple interactions transmitted across the network. Now the entity module notifies the federation only that a collision occurred, not resulting detailed state changes. Each entity computes those state changes locally as a result of the interaction. The result is a series of simulation actors whose behaviors and polygonal representations load dynamically at runtime. This allows simulation managers to easily experiment with the content of the environment by adding and subtracting functionality at runtime. The tendency is to think that this applies only to the graphically represented entities but it could mean that data loggers or analysis modules dynamically load and unload to collect and analyze simulation data. Bamboo provides an unprecedented method of adding functionality to an executing networked virtual environment.

3. Graphics Rendering

Bamboo's Visual module renders the graphical objects in the scene. The amHLAAdmin module and the entity modules update the object's position and orientation.

23

Each entity module registers callbacks with the Visual module to ensure accurate rendering of the simulation entity. These callbacks call the appropriate functions when the system needs to render the graphical representation of each entity. See Figure 4 for the callback tree representing this implementation.

B.   FEDERATION OBJECT MODEL/SIMULATION OBJECT MODEL

This implementation has a simple FOM. Table 4 displays the FOM tables for the implementation. The FOM and SOM are the same for this implementation because each federate is based on the amHLAAdmin Module common for all federates.   The FOM defines the Entity State Object and its attributes that define the state of the object. The Entity State Object defines the attributes that will be communicated to the network. Do not confuse this object with the software objects defined in the implementation. Those C++ objects may define more variables that define their state but are not communicated to the network.   The FOM also defines a very simple interaction called Collision. Its only parameter is the ObjectID number assigned to the entity that has been collided with. The purpose of this interaction is to communicate to the federation the ObjectID of the entity that was damaged. Each federate then updates the state of that entity.

Object Class Structure Table

EntityState (PS)

Object Interaction Table

Interaction Structure	Initiating Object		Receiving Object		Interaction Parameters	Init Sense React
	Class	Affected Attributes	Class	Affected Attributes		
Collision	EntityState	None	EntityState	Damage	EntityID	IR

Attribute Parameter Definition Table

Object/Interactio n	Attribute Parameter	Datatype	Card-inalit y	Units	Reso-lution	Accuracy	Accuracy Condition	Update Type	Update Condition	Transferabl e Acceptable	Updateabl e Reflectabl e
EntityState	Position	any	1			perfect	always	Periodic		TA	UR
	Orientation	any	1			perfect	always	Periodic		TA	UR
	Velocity	any	1			perfect	always	Periodic		TA	UR
	Ammunition	any	1			perfect	always	Periodic		TA	UR
	Damage	any	1			perfect	always	Periodic		TA	UR
Collision	EntityID	unsigned long	1			perfect	always	N/A	N/A	N/A	N/A

Table 4: Implementation FOM Tables

C. IMPLEMENTATION CONCLUSION

Recall the goal is a dynamically extensible, flexible, consistent, and specific NVE. Dynamic extensibility is accomplished by using Bamboo to load and unload modules. The NVE is consistent because the amHLAAdmin module ensures all entity and terrain modules required by the designers be loaded when required. Each module is specific because the programmer is only concerned with one module representing a particular entity. That programmer no longer has to concern himself with the representations and behaviors of the other entities in the environment. Finally, all of this adds up to flexible environment that can be changed during runtime to the state required by the simulation managers.

The preceding sections provide the overview of how the implementation accomplishes the goal of providing a dynamically extensible, flexible, specific and consistent NVE. Appendix A provides a detailed implementation tutorial that guides a user through the details required to implement an amEntity module. Appendix B provides the code examples that accompany the implementation tutorial. Together these two documents walk a user through the correct use of the HLA RTI, Bamboo and the amHLAAdmin module.

V. RESULTS AND LIMITATIONS

A. PERFORMANCE RESULTS

Two measures of performance were used to judge the merits of this implementation: frame rate per second and average time to clear the event buffer. Frame rate per second describes how the implementation impacts the graphics subsystem. Falling below ten frames per second severely hampers virtual environment realism. Time to clear buffers shows the impact of increasing numbers of entities on the system's ability to update each one. Each measure was sampled with different communication reliability parameters, specific numbers of entities, and federates participating in the environment.

1. Test Description

The system used to evaluate the implementation is a Windows NT 4.0 LAN, with a 166 MHz Pentium MMX CPU and 32 Megabytes of RAM. Each system connects to the 100 Mbps LAN with a 3Com 10/100 Mbps network interface card through a 100 Mbps hub. The graphics subsystem on these systems is not accelerated. All graphics are computed on the system processor. There was a single federate per system. Each federate computed and transmitted position, orientation, and velocity data every fifth frame.
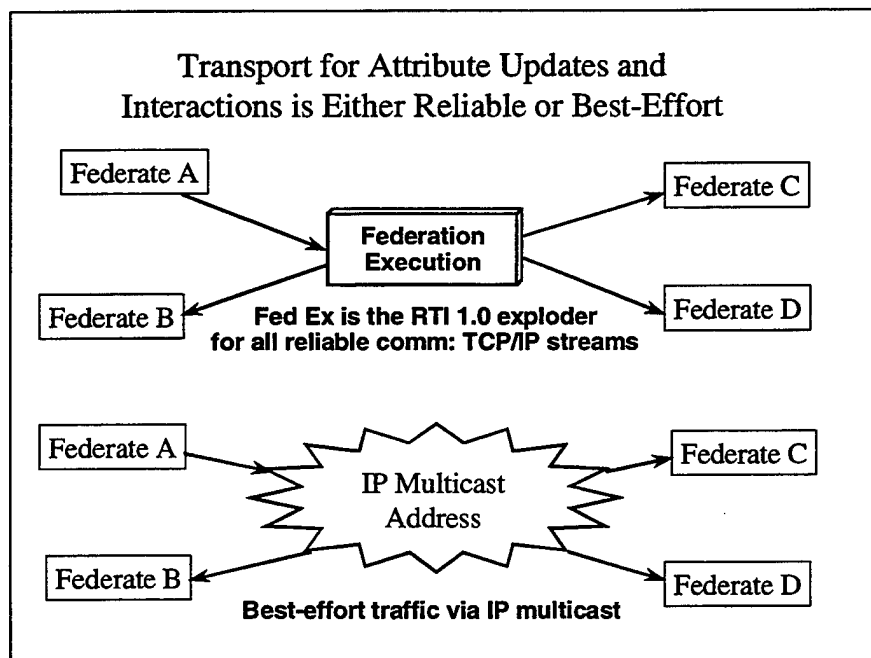
Figure 9: HLA Message Transport Types

There are two communication reliability settings defined by the HLA IF Spec: FED_RELIABLE and RED_BEST_EFFORT. FED_RELIABLE uses the federation execution to ensure that each message of this type is delivered to each federate in the federation. This adds a significant number of messages to the network since acknowledgements are required to confirm delivery of each message. FED_BEST_EFFORT reduces message traffic compared to FED_RELIABLE because each federate transmits its messages to a multicast address where every other federate reads the messages. There is no requirement for acknowledgements, but there is a small chance that a message may be delivered improperly or not at all. Figure 9shows graphically the differences between FED_RELIABLE and FED_BEST_EFFORT reliability settings. FED_RELIABLE transport creates a bottleneck because the federation execution process acts as a server ensuring that all federates receive each update. The FED_RELIABLE transport type only allowed two federates in the federation before the federation execution was bottlenecked, so no further measurements were taken. Two federates does not make a very interesting virtual world.

2. Frame Rate Results

Figure 10 shows the frame rate results for a federate using the FED_BEST_EFFORT transport protocol. Notice that the frame falls as the number of entities in the federation increases. This is to be expected. Each new federate brings 44 more polygons to the virtual world.

Average Frame Rate

	11	22	33	44	55	66	77
Frms/Sec	33.27	32.99	32.64	25.06	24.76	21.67	18.86

Number of Entities

Average Time to Clear Buffer

	11	22	33	44	55	66	77
Time	3.02	4.77	10.52	12.9	15.87	20.73	35.36

Number of Entities
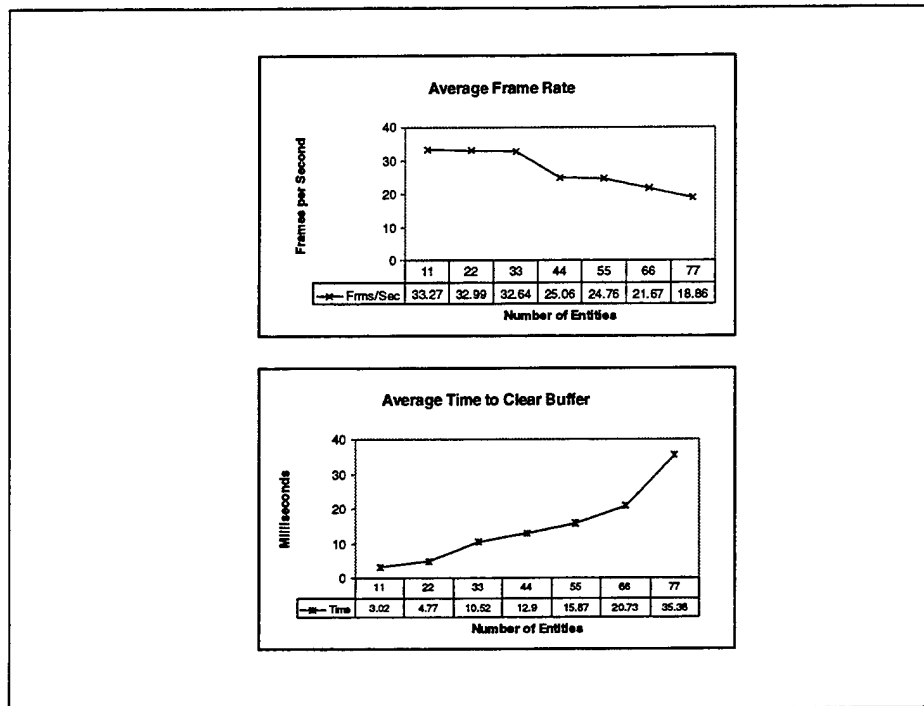
Figure 10: Performance Results

3.   RTI Event Buffer Read Results

Figure 10 shows the time to clear buffer results. The time to clear buffer is the time required to process all events in the RTI event queue. In this implementation all events are attribute updates. This test shows the average time it took to clear the event buffer. Recall that the reliable transport type was only able to accommodate two federates with a total of 22 entities in the federation, while the best effort transport type accommodated more than three times that with 7 federates and 77 entities. Notice that the average time to clear the buffer increases with the number of entities in the federation. This could limit scalability, but recall the limited power of the test systems and that the entity module in the test does not implement dead reckoning. Dead reckoning could reduce the number of entity updates thus improving the time to clear the buffer.

B.   LIMITATIONS

There are two major limitations in this implementation. The RTI limits the implementation due to its static nature. The design of the amHLAAdmin module is limited because it lacks a method of extending the executable by providing a separate thread to handle entity updates.

29

1. HLA/RTI Generalization

This implementation is generalized except for the HLA/RTI functionality. The RTI limits the functionality in two ways. First, the RTI's use of text files and environment variables limit the flexibility of this implementation. An API interface that set the RTI's state would significantly increase the flexibility of the system. Implementation of the HLA functionality requires that RTI be installed on every machine it is used on. It would be much more flexible if the RTI could be instantiated anytime and its state set with API function calls. This would allow the RTI to be part of the amHLAAdmin module and could be loaded in one step without having to actually install the RTI so that the static text files and environment variables are defined. Second, the FOM is predefined and parsed by the RTI at runtime. If there was an API interface to change the FOM during runtime, the federation could extend its capability without requiring that the federation execution be restarted.

2. HLA Memory Footprint

Each federation has three processes that must exist in order for the federation to operate: the RTI executive process, the federation executive, and at least one federate. The RTI executive process requires approximately 2.5 MB of RAM in order to run, while the federation executive requires 2.5 MB. Each federate's memory requirements will vary, but the local RTI component, or RTI ambassador requires 12 MB. This is a total of 17 MB required on one system. This is a significant amount of memory that must be considered when designing systems portable to desktop systems.

3. amHLAAdmin Module Limitations

The amHLAAdmin module does not provide a method for extending the executable. If this module provided a set of callbacks similar to the visual module's app, cull, draw system, the implementation could better control the sending and receiving of information through the RTI. For example, a separate thread could be started that continuously "ticks" the RTI, thus keeping its receive buffers clear. Additionally, a generalized system that allows each module to register callbacks on an update loop would provide a more efficient method of updating entity state. These improvements were not made because the significance of the processing time required to update entity state was not discovered until the final performance tests were run. Future improvements to the system will require a better method of managing processor time for the RTI to update entity state.

VI. CONCLUSIONS

A.   CONCLUSION

The implementation meets the requirements set out in the introduction concerning the improvements NVEs require. The requirements are the virtual environment must be dynamically extensible, flexible, specific, and consistent. This implementation is dynamically extensible. Bamboo does an excellent job of providing this capability by implementing the ability to dynamically extend the executable and providing a system that defines module dependency. The ability to load and unload modules on the command of the user or the system at any time during execution adds flexibility to the NVE. This means simulation designers can rapidly prototype simulation executions in real time and effectively design and implement the virtual environment. This demonstration verifies that simulation designers can be more specific. In other words, programmers just need to implement their module. They no longer need to represent the other entities that will exist in the NVE or approximate their behavior. An entity module that can be loaded as the situation changes and unloaded when no longer needed represents these remote entities. Finally, this system ensures that all simulations operating in the NVE are consistent with each other. The problem of different terrain models at each simulation site is solved because all sites receive the terrain module from the same location and execute it in the same manner. This same logic applies to the entity modules.

B.   FUTURE WORK

The following lists future projects that could improve real time virtual environments.

1.   Network Bandwidth and Latency of the RTI.

It is very difficult to quantify the effects the RTI has on the network. We do not know the methods used to keep the distributed local RTI components updated. What is the time management method employed by the RTI? How are the receive buffers filled and emptied? What is the most efficient method of clearing the buffers? The results indicate that more information is required to increase the numbers of entities modeled in a real time virtual environment.

31

2. Methods for Reducing Network Traffic Required to Maintain Consistent Entity State.

This work refers to ways to improve the RTI for real-time simulations. Area of interest management is a method of accomplishing this goal. The new version of the RTI, version 1.3-3, has implemented its own area of interest management system call Data Distribution Management (DDM). Future work could entail implementing support for DDM into this implementation.

3. Improvements to the Current Implementation

There are several ways to improve this implementation. First, the amHLAAdmin module could implement a series of callback handlers that accomplish the entity updates and interactions required by the system. This improvement would make the implementation more general by removing the requirement for the amObject pure virtual class interface. Next, multi-thread the implementation to provide specific amounts of processing time to the RTI and visual module. Finally, implement a system that ensures that locally computed objects are updated first. Currently, all entities are in the same list. Locally computed objects should have a separate list so they may be updated at a greater rate.

APPENDIX A: IMPLEMENTATION TUTORIAL

A. INTRODUCTION

This tutorial describes how to implement a module that will operate using the HLAAdmin implementation. It has three sections: Bamboo module implementation, HLA Implementation, and Demo. The Bamboo module implementation section describes how to build a module for use with the Bamboo virtual environment toolkit. The HLA implementation section outlines the HLA concepts that the user must understand to implement a module for the HLAAdmin implementation. Finally, the Demo section describes how to run the existing implementation. Taking each section in turn will ensure the user of an overall understanding of the HLAAdmin implementation.

The system requirements for this implementation are, Windows NT 4.0, Visual C++ 5.0, RTI 1.0-3 and rktools. Visual C++ is the compiler used in the examples and rktools provides the functionality needed to use makefiles that are very similar to UNIX makefiles. All the RTI functionality described in this tutorial is explained in detail in the RTI Programmers Guide, see the bibliography in the main section of the thesis.

B. BAMBOO MODULE IMPLEMENTATION

The easiest way to grasp how to implement a Bamboo module is to use one as an example. I will use the amBoid module, available in the code appendix, as the example module. Bamboo modules are made up of a minimum of four files: module.h, module.c, amBoid.h, amBoid.c. (The amBoid files are examples; any name can be there.) Each module is identified to the Bamboo kernel by six global functions defined in the module.c file: getModuleName(), getModuleVersion(), getModuleDate(), getModuleText(), initModule() and exitModule(). As a module is loaded, Bamboo creates a bbModule object that holds pointers to these functions.

When the module is loaded, the bbModule object executes the initModule() function. The initModule() function does the work required to add the module to the executable and instantiate any object required by the module. The exitModule() function removes the structures used to integrate the module into the executable and deallocates

33

the memory associated with any objects created for use by the module being deleted. Notice in the example module.c that the definition for the initModule() function makes one call to the initamBoid() function in amBoid.c, and the exitModule function makes one call to the exitamBoid() function in amBoid.c.

The initamBoid() function does the functions described above. This function is run only once immediatley after the module is loaded into memory. First, it instantiates a Boid object for use by the module. Then it calls the initKeyboardFunc() to add callbacks for keyboard events. This is one way to add the module into the program execution. The other method is to add callbacks that will include the module's functionality into the executable. This module is added to the executable by adding callbacks to the Visual module. The initVisualModule() function adds a callback to the preapp callback handler. The callback is associated with the preAppFunc() defined in amBoid.c. This function defines the keyboard controls and behavior for the Boid object created in the loadBoid() function. After this function is executed, the module executes as part of the Bamboo.exe executable until a command to unload the module.

On the command to unload a module, the bbModule object calls the exitModule() function defined in the module.c file. This function in turn calls the exitamBoid() function. The exitamBoid() does the housekeeping required to remove from memory all structures like callbacks or objects related to the module. In this case, the exitamBoid() function first removes the event responses from the keyboard. Notice that these commands are in the reverse order of the commands that were used to create the keyboard events. Next the preapp callback is removed. Finally, all objects associated with this module are deleted from memory.

The last Bamboo feature that will be discussed deals with module dependency. Bamboo extends the plug-in metaphor by implementing a system that allows modules to depend on other modules for execution. In this example, the amBoid module depends on the npsVisualModule, the bbKeyboardModule and the amHLAAdmin module. The module.txt file defines these dependencies. Bamboo ensures that all dependent modules are loaded first before the module that needs them is loaded.

34

This concludes the Bamboo section. It is not complicated to load and unload a Bamboo module. The user is required only to define the six functions in the module.c file. Understand that Bamboo's greatest strength is its convention that defines generalized methods to load and unload modules while providing a system that ensures module dependencies are enforced.

C.   HLA IMPLEMENTATION SECTION

HLA is implemented mainly in the amHLAAdmin module. Entity modules just make an Admin API call to pass or receive information to the RTI. There is one exception to this that I will address later in this section. The Admin API wraps ups all RTI ambassador functions. When the amHLAAdmin module loads, it instantiates a single object of type Admin. The entire API is static functions defined in the Admin class.

The Admin class uses the following RTI functionality (* is a Federate Ambassador function):

        Federation Management
                CreateFederationExecution
                JoinFederationExecution
                ResignFederationExecution
                DestroyFederationExecution
        Declaration Management
                PublishObjectclass
                PublishInteractionClass
                SubscribeObjectClassAttributes
                SubscribeInteractionClass
        Object Management
                RequestID
                RegisterObject
                UpdateAttributeValues
                DiscoverObject*
                ReflectAttributeValues*
                SendInteraction
                ReceiveInteraction*
                DeleteObject
                RemoveObject*
        RTI Services
                Tick

35

The RTI has other functions that do not apply to this implementation like time management and ownership management. If this functionality is required, the Admin object can be extended. Refer to the Admin.c file in the code appendix for the following functions. Notice the Admin class constructor is protected. There can be only one instance of the Admin class active so a singleton is implemented called getInstance() that creates the object the first time it is called and returns the pointer to the object every time it is called. The Admin constructor instantiates the RTI ambassador (rtiAmb) and the Federate ambassador (fedAmb) objects. The rtiAmb is the predefined object that contains all the functionality that implements the IF Spec. The fedAmb is a pure virtual class that defines the interface between the RTI and the implementation. Each rtiAmb call results in a return value from the RTI or, in the case of updates, a call to a fedAmb function. Let us discuss how the implementation uses each of the above RTI services and discuss the code that implements the service.

1.   Federation Management

The Admin constructor shown in Admin.c. makes calls to Admin::createFederationExecution() and Admin::joinFederation(). The createFederationExecution() function calls Admin::rtiAmb->createFederationExecution(fedExecName) and passes it a char* that defines the name of the Federation execution. If the federation already exists, the rtiAmb throws a federation already exists exception. If the federation is new, then a fedex process is spawned by the RTI. The Admin::rtiAmb->joinFederationExecution(federateName, fedExecName, fedAmb) passes it the name of the federate, the federation execution name and a reference to the fedAmb object. The name of the federate is name mangled by the RTI so it does not have to be unique and the fedExecName must be the same as was used in the createFederationExecution() function. The fedAmb reference is a pointer to the fedAmb created in the Admin constructor. The loop is used to give to the federate multiple tries to join the federation because the federation execution may require more time to configure itself before it is able to return the federate ID. The federateID is a unique number

assigned by the fedex that identifies your federate. It is not required anywhere but may be required by the user.

The Admin::resignFederate() function handles the resign federate and destroy federate management functions. This function first calls Admin::rtiAmb->resignFederation() and passes an enumeration that defines the clean up the user wants done prior to federate resignation. The implementation uses DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES. This value results in removeObject() calls to the fedAmb for all locally updated entities and releases the attributes on any objects that must transfer attribute ownership because the federate is resigning. The final call in this function is to the Admin::rtiAmb->destroyFederation(). This function destroys the federation if there are no other federates operating, otherwise an exception is thrown and the federation continues.

2.   Declaration Management

Declaration Management identifies to the rtiAmb all the objects/interactions and attributes/parameters that the federate is interested in. After the federate is joined, the user must get from the rtiAmb the enumerated values for the objects/interactions and attributes/parameters computed when the FOM was parsed. The Admin::Init() function uses RTI support functions that use the information from the FOM to provide these enumerated values for the different objects/interactions and attributes/parameters. Notice the Init() function uses specific functions that provide the enumerated values for the different Objects (getObjectClassHandle(char *)) and Attributes (getAttributeHandle(ObjectTypeEnum,char *)) that will be used by the federate. The char * parameter must match exactly with the strings used to describe the Objects and Attributes in the FOM.

After the enumerated values are saved by the user using the Init() function, the user calls the Admin::PublishAndSubscribe() function. This function makes RTI calls that tell the rtiAmb which Objects/Interactions and Attributes/Parameters the federate will publish and subscribe. The mechanics of this operation begin with the user building an RTI::AtrributeHandleSet. This data structure holds the attribute enumerations for a

specific object. The first part of the PublishAndSubscribe() function builds the RTI::AtrributeHandleSet. First the set is declared. Then space is allocated for five attributes using the create method of the RTI::HandleSetFactory object. Finally, all attributes that the federate wants are added to the set using the add(AttributeHandle) method and passed the enumeration for the specific attribute to be added. After all the attributes are added to the HandleSet the Admin::ms_rtiAmb->subscribeObjectClassAttribute(ClassHandle, *HandleSetFactory) and Admin::ms_rtiAmb->publishObjectClass(ClassHandle, *HandleSetFactory) are called to tell the rtiAmb those objects that will be published and subscribed. The last functions are for subscribing and publishing Interactions. Notice that these functions do not require a HandleSetFactory. When a federate subscribes or publishes an interaction, it accepts responsibility for all the parameters of that interaction, hence there is no need to tell the rtiAmb which parameters it is responsible for.

The above functions accomplish all the tasks required to publish and subscribe to objects/interactions and attributes/parameters. These are required tasks that all federates must accomplish in order to participate in the federation.

3.   Object Management

Object management services provide the functionality to identify entities to the RTI, discover them on remote federates, and update their attributes during the federation execution. The RTI requires that all entities are associated with an Object defined in the FOM. All entities must possess a unique ID number provided by the RTI. This number identifies the entity on all federates in the federation and ensures updates and interactions are processed on the correct entity.

The first task when adding an entity to the federation is getting its ObjectId from the rtiAmb then registering that ID with the rtiAmb. The Admin::registerObject() function accomplishes these tasks. First the Admin::ms_rtiAmb->requestID(RTI::ObjectIDCount, RTI::ObjectId, RTI::ObjectId) function provides the ObjectID numbers. Then the Admin::ms_rtiAmb->registerObject(RTI::ClassHandle, RTI::ObjectId) function registers

38

the entity with the rtiAmb and associates it with a Object that was previously published or subcribed.

After the object is registered with the rtiAmb, the entity will be updated and its attributes reflected on all participating federates. This task requires that the entity be discovered by each federate in the federation. Discovery requires that the entity be updated at least once. On the first update, the rtiAmb calls the FederateAmbassador::discoverObject() function. This function then ensures the module is loaded that represents this object. After the module is loaded an entity is instanced and its state is updated with the attributes passed by the rtiAmb. Refer to the AdminFederateAmbassador.c file for the discoverObject() function. After the object is discovered, all updates come to it through the FederateAmbassador::reflectAttributeValues() function. We will look at how the attributes are updated first. Then we will look at the discoverObject() function is detail.

Entities are updated using the rtiAmb->updateAttributeValues() function. The implementation calls this function in the Admin::sendEntityUpdate(RTI::ObjectID , RTI::AttributeHandleValuePairSet& , const RTI::UserSuppliedTag). Each entity must implement the virtual function sendUpdates() defined in amObject. This function produces a RTI::HandleValuePairSet then calls the Admin::sendEntityUpdates() function. Refer to boid.c in the code appendix for listings of entity functions. Boid::sendUpdates() first creates a HandleValuePairSet with the attributes in it that it wants to update. Then it calls Admin::sendEntityUpdate() and passes it the ObjectId of the entity to be updated, its HandleValuePairSet and the UserSuppliedTag which identifies the module that the entity is modeled by. Admin::sendEntityUpdates() then calls the Admin::rtiAmb-> updateAttributeValues(ObjectId, HandleValuePairSet, FederationTime, UserSuppliedTag( ). This command sends a packet out on the network containing the data specified. When a remote federate receives the data the rtiAmb calls the FederateAmbassador::discoverObject() function if the ObjectId is not known to the federate or the FederateAmbassador::reflectAttributeValues() function if the entity already exists in that federate.

39

The AdminFederateAmbassador:: reflectAttributeValues() function calls the Admin::receiveUpdate() function. This function searches the list of entities and then calls the Boid::receiveUpdates() virtual function defined by the amObject class. The Boid::receiveUpdates() function decodes the HandleValuePair using the getValue() method and updates the appropriate state variable. It then deletes the HandleValuePair.

If the entity is not known, the rtiAmb calls the FederateAmbassador ::discoverObject() function. This function ensures the appropriate module is loaded then calls Admin::addSimEntity() to ensure the entity is added to the list of entities displayed by the federate. On the next update, the entity's state is updated using the previously described process.

The process for interactions is very similar. An entity generates an interaction and uses its Boid::sendInteraction() function. This function calls the Admin::rtiAmb->sendInteraction() function. This provides more flexibility to the federate. The interaction is received on the remote federates and the rtiAmb calls FederateAmbassador::receiveInteraction() function. This function then calls the Admin::receiveInteraction() function which finds the affected entity and calls the Boid::receiveInteraction() function. In this function the parameters changed by the interaction are modified and the ParameterHandleValueSet is deleted.

This implementation also has the ability to delete entities from the federation. This is accomplished using the RTI functions deleteObject and removeObject. When an entity is identified for deletion, the entity is deleted locally then the rtiAmb is notified of the deletion through the deleteObject function. This causes a network message that results in the remote federate rtiAmbs calling the FederateAmbassador::removeObject() function. The details follow for this implementation. First an entity identified for deletion and its geometry are deleted from the local environment. Then the Admin::removeAmObject() function is called. This function removes the entity from the object list and calls Admin::rtiAmb->deleteObject() which results in a call to the FederateAmbassador::removeObject() function. This function then calls the Admin::removeAmObject() on the remote federate. This process notifies the federation of

the entities to be deleted and makes the calls necessary to remove the entity from the object lists in all federates.

4.   RTI Services

The final topic concerns how the rtiAmb is allocated processing time. The RTI is a single threaded application, so processing time is allocated through the use of the tick() command. This command causes the rtiAmb to accomplish tasks such as clearing buffers and executing callbacks based on the status of the federation.

The tick() function has two forms. The form used in the implementation ticks the rtiAmb once for each call. The other form , tick(min,max), will tick the rtiAmb for a specific length of time. Both forms return a boolean value indicating if there are more events in the queue. This function must be run periodically in order for the federation to operate. Waiting too long to tick can result in overfull buffers that take an inordinate amount of time to clear thus reducing the overall speed of the federation. In this implementation the rtiAmb is ticked once per frame to provide the most updated information to each federate.

D.   DEMO INSTRUCTIONS

❖ Start the RTI executive.
❖ Open two command windows.
❖ In both command windows, change directory to the bamboo directory.
❖ Type   bamboo amHLAAdmin   in both windows.
  ➢ CTRL-E exits amHLAAdmin (mouse must be in the execution window)
  ➢ The federation execution (fedex) should have started in another window
  ➢ There should be two light green windows (move the top one to see the other)
❖ Now add modules to the federation.
  ➢ Add the Arena Terrain.
    ▪ Type   CTRL-L   to load module.
    ▪ Enter the module name   amArena.
    ▪ Type   CTRL-T   to promulgate the module to all federates.
❖ Now populate the environment.

41

- ➢ Type   CTRL-L   to load module.
- ➢ Enter the module name   amBoid.
- ➢ Type   B   to create the Boid.
  - ▪ Control the boid with the arrow keys.
  - ▪ The A key makes it go forward.
  - ▪ The S key makes it go backward.
- ❖ Do the above in both windows.
  - ➢ Turn the boid that you can the other in both windows.
  - ➢ Collide one boid with the other.
    - ▪ This produces an interaction.
    - ▪ After 10 hits the boid is destroyed and should disappear.
    - ▪ To add another boid type B again.

APPENDIX B: IMPLEMENTATION CODE LISTINGS

Table of Contents for the Code Listings

amHLAAdmin Files

```
Bamboo1.0b
4
npsVisualModule
bbKeyboardModule
npsFlyingCameraModule
amPageModule
```

```
// *****************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: module.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: This file defines the global functions required by
//     every dynamically-linked library.  How these functions are
//     implemented is arbitrary, but it is useful have RCS
//     automate some of the work.
//
// September 1998, Masters Thesis
// *****************************************************************


#ifndef __module
#define __module

// *****************************************************************
//    FUNCTION PROTOTYPE SPECIFICATIONS
// *****************************************************************

#ifdef __cplusplus
extern "C" {
#endif

  SIM_API const char *getModuleName();
  SIM_API float getModuleVersion();
  SIM_API const char *getModuleDate();
  SIM_API const char *getModuleText();
  SIM_API bool initModule();
  SIM_API bool exitModule();

#ifdef __cplusplus
};
#endif

#endif // __module
```

```
// *****************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: module.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: This file defines the global functions required by
//     every dynamically-linked library.  How these functions are
//     implemented is arbitrary, but it is useful have RCS
//     automate some of the work.
//
// September 1998, Masters Thesis
// *****************************************************************
// *****************************************************************
//    INCLUDES AND EXTERNS
// *****************************************************************
#include <stdlib.h> // atof
#include "module.h"
#include "amHLAAdmin.h"


// *****************************************************************
//    CODE
// *****************************************************************
const char *getModuleName()
  {
  return "amHLAAdmin";
  }

float getModuleVersion()
  {
  return 1.0;
  }

const char *getModuleDate()
  {
  return "1998/08/01 06:05:48";
  }

const char *getModuleText()
  {
  return "amHLAAdmin Module -- CNTL-E to exit";
  }

bool initModule()
  {
  initAdmin();
  return 1;
  }

bool exitModule()
  {
  exitAdmin();
  return 1;
  }
```

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••
//   EXECUTIVE SUMMARY
//
// File Name: amHLAAdmin.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Defines the module controls for the amHLAAdmin module.
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••


#ifndef _amHLAAdmin
#define _amHLAAdmin


// ••••••••••••••••••••••••••••••••••••••••••••••••••
//   INCLUDES AND EXTERNS
// ••••••••••••••••••••••••••••••••••••••••••••••••••
#include "admin.h"

// ••••••••••••••••••••••••••••••••••••••••••••••••••
//   FUNCTION PROTOTYPE SPECIFICATIONS
// ••••••••••••••••••••••••••••••••••••••••••••••••••

void initAdmin();

void exitAdmin();

// ••••••••••••••••••••••••••••••••••••••••••••••••••
//   INLINED MEMBER FUNCTIONS
// ••••••••••••••••••••••••••••••••••••••••••••••••••


#endif // _bbHLAAdmin
```

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
//   EXECUTIVE SUMMARY
//
// File Name: amHLAAdmin.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Defines the module controls for the amHLAAdmin module.
//     This module manages the communication between federates and
//     has data structures avilable to facilite collision detection
//     and other tasks that require comparison with all other
//     simulation entities.
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
//        INCLUDES AND EXTERNS
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

#include "Rti.hh"
#include "amHLAAdmin.h"
#include "bbKeyboard.h"
#include "bbEventResponse.h"
#include "bbCallback.h"
#include "bbModule.h"
#include "admin.h"
#include "npsWindow.h"
#include "npsViewport.h"
#include "npsCamera.h"
#include "npsVisual.h"
#include "bbMouse.h"
#include "amObject.h"
#include "ace/OS.h"
#include <math.h>
#include <GL/gl.h>

#ifdef SGI
  #include "bbSGI.h"
#endif

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
//        CODE
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••


// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: initAdmin()
// Task:  Makes initial calls to make module a part of the executable
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void initAdmin(){
    void initDisplayWindow();
    void initKeyboardModule();
    void createMainCallbackHandler();

    Admin::getInstance();
```

```
    createMainCallbackHandler();
    initKeyboardModule();
    initDisplayWindow();

}// end initAdmin

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: exitAdmin()
// Task:  deletes callbacks etc to ensure the module is unloaded
//    gracefully

// Return Value: void

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void exitAdmin(){
    bbCallback*         callback;
    bbEventResponse*    eventResponse;
    bbCallbackHandler*  callbackHandler;

    Admin::getMutex()->acquire();
    // delete keyboard callbacks
    // exit key
    eventResponse = bbEventResponse::findObject("amHLAAdminER_Exit");
    callback = bbCallback::findObject("amHLAAdmin_Exit");
    eventResponse->removeCallback(callback);
    delete callback;

    // resign federate
    Admin::resignFederate();

    // delete Admin instance
    Admin* tmp = Admin::getInstance();
    delete tmp;

    Admin::getMutex()->release();

    exit(0);

}

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: initDisplayWindow()
// Task:  inits main window for the federate.  Bamboo currently only
//    allows on window open at time.  This is the window for the
//    federate
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void initDisplayWindow(){

    void preDrawFunc(bbObject *object, bbData *data);

    npsWindow        *window;
```

```
    npsCamera          *camera;
    npsViewport        *viewport;
    npsVec3f           position;
    bbCallback         *callback;
    bbCallbackHandler  *callbackHandler;

    //create the window for the simulation
    window = new npsWindow(370, 370);
    window->setName("AdminWindow");
    viewport = new npsViewport(0.0f, 1.0f, 0.0f, 1.0f);
    viewport->setName("AdminViewport");

    // camera to view the area
    camera = new npsCamera();
    camera->setName("AdminCamera");
    camera->setFarClip(200.0f);
    position.set(0.0f, 3.0f, 10.0f);
    camera->setPosition(position);
    camera->setClearColor(0.66f, 1.0f, 0.66f, 1.0f);
    camera->setAspectMode(npsCamera::CALC_VERT);
    viewport->setCamera(camera);

    window->addViewport(viewport);

    // callback to change position of camera and update screen
    callback = npsVisual::drawCallback();
    callbackHandler = callback->getPreCallbackHandler();
    callback = new bbCallback();
    callback->setName("amHLAAdmin_preDraw");
    callback->setFunc(preDrawFunc);
    callbackHandler->addCallbackLast(callback);


}//initDisplayWindow

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: preDrawFunc()
// Task:  calls Admin::display to update simulation entities
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void preDrawFunc(bbObject *object, bbData *data){

    Admin::getInstance()->display();

}// end preDrawFunc

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: createMainCallbackHandler()
// Task:  puts callback on the bamboo main event loop.  this is done
//    because RTI 1.0-3 requires that the RTI be ticked from the
//    same thread that instantiated it
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
```

```c
void createMainCallbackHandler() {
    void hlaTickRTI(bbObject *object, bbData *data);

    bbCallback*            callback;
    bbCallbackHandler*     callbackHandler;

    callbackHandler = bbCallbackHandler::findObject("mainCallbackHandler");
    callback = new bbCallback();
    callback->setName("rtiTick");
    callback->setFunc(hlaTickRTI);
    callbackHandler->addCallbackLast(callback);

} // end createMainCallbackHandler()


// ************************************************************
// Function Name: initKeyboardModule
// Task:    creates module keyboard callbacks
// Return Value: void
// ************************************************************

void initKeyboardModule() {

    void exitFunc(bbObject *object, bbData *data);
    void hlaUpdateBoid(bbObject *object, bbData *data);
    void hlaTickRTI(bbObject *object, bbData *data);
    void hlaUpdateObject(bbObject *object, bbData *data);
    void hlaUpdateClass(bbObject *object, bbData *data);

    bbKeyboard              *keyboard;
    bbEventResponse         *eventResponse;
    bbCallback              *callback;

    // get the keyboard device
    keyboard = bbKeyboard::getInstance();

    // set up exit key
    eventResponse = new
      bbEventResponse(bbKeyboard::KEY_E | bbKeyboard::CTRL_MASK |
bbKeyboard::DOWN_TRANS);
    eventResponse->setName("amHLAAdminER_Exit");
    callback = new bbCallback();
    callback->setName("amHLAAdmin_Exit");
    callback->setFunc(exitFunc);
    eventResponse->addCallbackLast(callback);
    keyboard->addEventResponse(eventResponse);

}//end initKeyboardModule()

// ************************************************************
// Function Name: exitFunc
// Task:    callbck function for cntrl-e that exits the federation and
//          ensures all callbacks are cleaned up
// Return Value: void
// ************************************************************
```

```c
void exitFunc(bbObject *object, bbData *data){
    int mouseInWindow();

    bbModule *module;

    if (mouseInWindow()){

        module = bbModule::findObject("amHLAAdmin");
        if (!module){
            cout << "amHLAAdmin is not currently loaded...ignoring" << endl;
            return;
        }//end if

        if (!bbModule::unload(module)){
            cout << "  Error unloading amHLAAdmin " << endl;
            cout << "  Fatal Error - aborting executable!" << endl << endl;
            exit(0);
        }// end if
    }//end if

}// exitFunc()


// ************************************************************
// Function Name: hlaTickRTI
// Task:    callback func that makes the RTI calls that tick the rti
//          this provides the cpu time for the rti to get its work done
// Return Value: void
// ************************************************************

void hlaTickRTI(bbObject *object, bbData *data){

    RTI::FederationTime tmpTime(1.0);

    while (Admin::getInstance()->tickRTI());

    Admin::getInstance()->updateFederation(tmpTime);

}// end hlaTickRTI()


// ************************************************************
// Function Name: mouseInWindow
// Task:    ensures the mouse pointer is the window that you want to
//          update
// Return Value: integer that is interpreted for use as a bool
// ************************************************************

int mouseInWindow() {
    int flag = 0;

    npsWindow               *window;
    npsViewport             *viewport;
    window = npsWindow::findObject("AdminWindow");
    viewport = npsViewport::findObject("AdminViewport");
    float          x,y;
    x = bbMouse::getX();
```

```c
    y = bbMouse::getY();
    bbScreen::normalizeVal(&x,&y);

    if (viewport->getWindow()->isValInside(x,y)){
        flag = 1;
    }
    return flag;
}// end mouseInWindow()


// end amHLAAdmin.c
```

```c
// ************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: admin.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Definition of Admin class
//     This file defines the utility and external API functions
//     required to manage an amCompatible federate
//
// September 1998, Masters Thesis
// ************************************************************

#include "Rti.hh"
#include "AdminFederateAmbassador.h"
#include "npsVec3f.h"
#include "npsGeometry.h"
#include "bbKeyboard.h"
#include "amObject.h"

#ifndef _admin
#define _admin


//defines


#ifndef SIM_API
    #ifdef VISUALCPP
      #define SIM_API __declspec(dllimport)
    #else
      #define SIM_API
    #endif
#endif


class SIM_API Admin
{

//member variables
public:

    // Simulation Entity data memebers
    static vector<amObject*>    ms_amObjectArray;
    static unsigned int    ms_NumberAmObjects;

    // Array of Objects used by modules
    static vector<amObject*> ms_ModuleArray;
    static unsigned int    ms_NumberModules;

    // The Terrain Object pointer
    static amObject* ms_Terrain;

    // time management variables
    static RTI::Boolean    ms_timeAdvGrant;
    static RTI::FederationTime  ms_grantTime;
```

```cpp
    static RTI::FederationTime   m_lastTime;

    // RTI Ambassador pointer
    static RTI::RTIambassador* ms_rtiAmb;

    static RTI::FederateHandle   m_federateID;


    // flag that signals whether interactions are valid
    static RTI::Boolean ms_InteractionFlag;

    // name of the Federation Execution
    static char* const m_FedExecName;

    // provides url to module locations
    static char* const ms_ModuleURL;


private:
    // Static Member Data
    // Federate Ambassador Pointer
    static AdminFederateAmbassador* ms_fedAmb;


    // RTTI data provided by the RTI during Init()
    static RTI::ObjectClassHandle       ms_AdminClassHandle;
    static RTI::AttributeHandle         ms_ModuleURLAttrHandle;
    static RTI::AttributeHandle         ms_ModuleVersionAttrHandle;
    static RTI::AttributeHandle         ms_ModuleNameAttrHandle;

    static RTI::ObjectClassHandle       ms_ESClassHandle;
    static RTI::AttributeHandle         ms_PositionAttrHandle;
    static RTI::AttributeHandle         ms_OrientationAttrHandle;
    static RTI::AttributeHandle         ms_VelocityAttrHandle;
    static RTI::AttributeHandle         ms_AmmunitionAttrHandle;
    static RTI::AttributeHandle         ms_DamageAttrHandle;

    static RTI::InteractionClassHandle  ms_CollisionInteractionHandle;
    static RTI::ParameterHandle         ms_EntityIdHandle;


    // Names for querying RTTI values

    static char* const          ms_ESClassStr;
    static char* const          ms_PositionStr;
    static char* const          ms_OrientationStr;
    static char* const          ms_VelocityStr;
    static char* const          ms_AmmunitionStr;
    static char* const          ms_DamageStr;
    static char* const          ms_CollisionInteractionStr;
    static char* const          ms_EntityIdStr;

    // constructor is private due to singleton
    Admin();

    // singleton
    static Admin *this_;
```

```cpp
    // mutex used to make Admin Thread Safe
    static ACE_Recursive_Thread_Mutex adminMutex;

// member functions
public:
    // destructor
    virtual ~Admin();

    // ensures update of the federation
    void updateFederation(RTI::FederationTime& newTime);

    // This is the Admin API portion

    // singleton getter
    static Admin *getInstance();
    // Send data to RTI
    static void sendCollisionInteraction(RTI::FederationTime theTime,
                               RTI::ObjectID oid);

    static void sendEntityUpdate(RTI::ObjectID theObject,
            RTI::AttributeHandleValuePairSet& theAttributes,
            const RTI::UserSuppliedTag          theTag);


    // Receive data from RTI
    static void receiveUpdate( RTI::ObjectID theObject,
            const RTI::AttributeHandleValuePairSet& theAttributes,
            RTI::FederationTime             theTime,
            const RTI::UserSuppliedTag          theTag,
            RTI::EventRetractionHandle          theHandle );

    static void receiveInteraction(RTI::InteractionClassHandle
            theInteraction,
            const RTI::ParameterHandleValuePairSet& theParameters,
            RTI::FederationTime theTime,
            const RTI::UserSuppliedTag theTag,
            RTI::EventRetractionHandle theHandle);

    // Manage Sim Entity Pointers
    static void display();
    static void addModuleFunctionPointer(amObject*);
    static amObject*       findSimEntity(RTI::ObjectID  objectId );
    static amObject*       findSimEntity(const char * moduleName);
    static RTI::ObjectID      registerObject();
    static amObject* addSimEntity(const char* moduleName, RTI::ObjectID);
    static amObject* checkEntityCollision(amObject* that);
    static void removeAmObject(RTI::ObjectID oid);

    // Module Object Management
    static RTI::Boolean moduleLoaded(const char* moduleName);
    static void loadModule(const char* moduleName);
    static void unloadModule(const char* moduleName);
    static amObject* findModule(const char* moduleName);
    static void removeModule(const char* moduleName);

    // RTI execution management
```

```cpp
    static RTI::Boolean tickRTI();
    static RTI::Boolean tickRTI(RTI::TickTime minTick,RTI::TickTime maxTick);
    static void advanceTime(RTI::FederationTime time);
    static void resignFederate();
    static void joinFederation();
    static void setTimeManagement();
    static void createFederationExecution();
    static void      Init( );
    static void      PublishAndSubscribe();
    static ACE_Recursive_Thread_Mutex* getMutex();


    // Accessor Methods

    // Static Accessor Methods

    static RTI::ObjectClassHandle       getESClassHandle()
            { return ms_ESClassHandle; };
    static RTI::AttributeHandle         getPositionAttrHandle()
                { return ms_PositionAttrHandle; };
    static RTI::AttributeHandle         getOrientationAttrHandle()
                { return ms_OrientationAttrHandle; };
    static RTI::AttributeHandle         getVelocityAttrHandle()
                { return ms_VelocityAttrHandle; };
    static RTI::AttributeHandle         getAmmunitionAttrHandle()
                { return ms_AmmunitionAttrHandle; };
    static RTI::AttributeHandle         getDamageAttrHandle()
                { return ms_DamageAttrHandle; };

    static RTI::InteractionClassHandle  getCollisionInteractionHandle()
            (return ms_CollisionInteractionHandle;)

    static RTI::AttributeHandle         getEntityIdHandle()
                { return ms_EntityIdHandle; };


}; // end Class Admin
#endif
```

```cpp
// ****************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: admin.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Method Definitions of Admin class
//    This file defines the utility and external API functions
//    required to manage an amCompatible federate
//
// September 1998, Masters Thesis
// ****************************************************************

#include "RTI.hh"
#include "admin.h"
#include "AdminFederateAmbassador.h"
#include "npsGeometry.h"
#include "bbKeyboard.h"
#include "bbModule.h"
#include "amObject.h"
#include "ace/OS.h"
#include "vector.h"


    // Static Variable initializations

    // singleton
    Admin *Admin::this_ = 0;

    RTI::RTIambassador*     Admin::ms_rtiAmb = NULL;

    AdminFederateAmbassador* Admin::ms_fedAmb = NULL;

    RTI::FederateHandle   Admin::m_federateID = 0;

    vector<amObject*>     Admin::ms_amObjectArray;

    unsigned int     Admin::ms_NumberAmObjects = 0;

    vector<amObject*>     Admin::ms_ModuleArray;

    unsigned int     Admin::ms_NumberModules = 0;

    amObject* Admin::ms_Terrain = NULL;


    //Time Management flags

    RTI::Boolean Admin::ms_timeAdvGrant = RTI::RTI_FALSE;
    RTI::FederationTime  Admin::ms_grantTime(1.0);
    RTI::FederationTime   Admin::m_lastTime;

    RTI::Boolean Admin::ms_InteractionFlag = RTI::RTI_FALSE;


    // RTTI data
```

```cpp
RTI::ObjectClassHandle        Admin::ms_ESClassHandle = 0;
RTI::AttributeHandle          Admin::ms_PositionAttrHandle = 0;
RTI::AttributeHandle          Admin::ms_OrientationAttrHandle = 0;
RTI::AttributeHandle          Admin::ms_VelocityAttrHandle = 0;
RTI::AttributeHandle          Admin::ms_AmmunitionAttrHandle = 0;
RTI::AttributeHandle          Admin::ms_DamageAttrHandle = 0;

RTI::InteractionClassHandle  Admin::ms_CollisionInteractionHandle = 0;
RTI::ParameterHandle         Admin::ms_EntityIdHandle = 0;

// Names for querying RTTI values
char* const        Admin::ms_ESClassStr = "EntityState";
char* const        Admin::ms_PositionStr = "Position";
char* const        Admin::ms_OrientationStr = "Orientation";
char* const        Admin::ms_VelocityStr = "Velocity";
char* const        Admin::ms_AmmunitionStr = "Ammunition";
char* const        Admin::ms_DamageStr = "Damage";

char* const        Admin::ms_CollisionInteractionStr = "Collision";
char* const        Admin::ms_EntityIdStr = "EntityID";

char* const        Admin::m_FedExecName = "HLA_Admin";

char* const        Admin::ms_ModuleURL = "URL not defined";

ACE_Recursive_Thread_Mutex   Admin::adminMutex;


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

// ****************************************************************
// Function Name: getInstance()
// Task:  singleton accessor function
// Return Value: Admin*
// ****************************************************************

Admin* Admin::getInstance(){

    if (!this_) {
        this_ = new Admin();
    }

    return this_;
}//end getInstance()


// ****************************************************************
// Function Name: Admin()
// Task:  constructor
// Return Value: none
// ****************************************************************

Admin::Admin() {
```

```cpp
    try // rtiAmb
    {
        Admin::ms_rtiAmb = new RTI::RTIambassador();
        Admin::ms_fedAmb = new AdminFederateAmbassador();

    } // try rtiAmb
    catch (RTI::ConcurrentAccessAttempted& e)
    {
        cerr << "Instantiate rtiAmb and fedAmb" << endl;
        cerr << &e << endl;
    }
    catch (RTI::Exception& e)
    {
        cerr << "Instantiate rtiAmb and fedAmb" << endl;
        cerr << &e << endl;
    }


    createFederationExecution();
    joinFederation();

}// end constructor


// ****************************************************************
// Function Name: ~Admin()
// Task:  destructor
// Return Value: none
// ****************************************************************

Admin::~Admin(){
    amObject* tmp;
    vector<amObject*>::iterator ptr;

    getMutex()->acquire();

    //delete all elements of Object Array
    ptr = ms_amObjectArray.begin();
    while(ptr != ms_amObjectArray.end()){
        tmp = (amObject*)*ptr;
        if(tmp){
            delete tmp;
        }//end if
        ptr++;
    }

    ms_amObjectArray.empty();

    //delete ms_ModuleArray
    ptr = ms_ModuleArray.begin();
    while(ptr != ms_ModuleArray.end()){
        tmp = (amObject*)*ptr;
        if(tmp){
            delete tmp;
        }//end if
```

```cpp
        ptr++;
    }

    ms_ModuleArray.empty();

    getMutex()->release();

    // delete ms_Terrain
    if(ms_Terrain) delete ms_Terrain;

    // delete the ambassadors
    if(ms_rtiAmb) delete ms_rtiAmb;
    if(ms_fedAmb) delete ms_fedAmb;
}


// ****************************************************************
// Function Name: findSimEntity( RTI::ObjectID objectId )
// Task:  find a simulation entity given its Object ID
// Return Value: amObject*
// ****************************************************************

amObject* Admin::findSimEntity( RTI::ObjectID objectId ){

    amObject* amPtr = NULL;
    amObject* tmp = NULL;

    vector<amObject*>::const_iterator   ptr;

    getMutex()->acquire();

    // iterate the vector ms_amObjectArray
    for (ptr = ms_amObjectArray.begin(); ptr != ms_amObjectArray.end();
ptr++){
        tmp = (amObject*)(*ptr);
        if(tmp && (tmp->getEntityID() == objectId)){
            amPtr = tmp;
            break;
        }//end if
    }//end for

    getMutex()->release();

    return amPtr;
}// end findSimEntity


// ****************************************************************
// Function Name: findSimEntity( const char* moduleName )
// Task:  find a simulation entity given its Module's name
// Return Value: amObject*
// ****************************************************************

amObject* Admin::findSimEntity( const char* moduleName ){

    amObject* amPtr = NULL;
    amObject* tmp = NULL;
```

```cpp
    vector<amObject*>::const_iterator   ptr;

    getMutex()->acquire();

    //iterate the vector
    for (ptr = ms_amObjectArray.begin(); ptr != ms_amObjectArray.end();
ptr++){
        tmp = (amObject*)(*ptr);
        if(tmp && strcmp(moduleName,tmp->getModuleName()) == 0){
            amPtr = tmp;
            break;
        }//end if
    }//end for

    getMutex()->release();

    return amPtr;
}// end findSimEntity

// ****************************************************************
// Function Name: registerObject()
// Task:  ask the RTI for a unique Object ID
//    register the Object as and Entity State Object in the RTI
// Return Value: RTI::ObjectID
// ****************************************************************

RTI::ObjectID Admin::registerObject(){

    RTI::ObjectID oid = 0;
    if ( Admin::ms_rtiAmb ){
        getMutex()->acquire();

        RTI::ObjectIDcount numObjects(1);

        // get the ID from the RTI
        try{
            Admin::ms_rtiAmb->requestID( numObjects, oid, oid );

        }
        catch (RTI::Exception& e)
        {
            cerr << "requestID" << endl;
            cerr << &e << endl;
        }

        // register the ID as an Entity State Class Object
        try{
            Admin::ms_rtiAmb->registerObject( Admin::getESClassHandle(), oid );

        }
        catch (RTI::Exception& e)
        {
            cerr << "registerObject" << endl;
            cerr << &e << endl;
        }
```

48

```cpp
        getMutex()->release();

    } // end if

    return oid;

}// end register object

// ************************************************************
// Function Name: Init( )
// Task:  requests the handle values for the user defined objects in
//    the FED file from the RTI
// Return Value: void
// ************************************************************

void Admin::Init( ){

    if ( Admin::ms_rtiAmb ){

        // RTTI for Objects
        try{
            Admin::ms_ESClassHandle = Admin::ms_rtiAmb-
>getObjectClassHandle(ms_ESClassStr);
        }
        catch (RTI::Exception& e){
            cerr << "getObjectClassHandle" << endl;
            cerr << &e << endl;
        }

        try{
            Admin::ms_PositionAttrHandle = Admin::ms_rtiAmb-
>getAttributeHandle(ms_PositionStr,
                                            ms_ESClassHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "getAttributeHandle" << endl;
            cerr << &e << endl;
        }
        try{
            Admin::ms_OrientationAttrHandle = Admin::ms_rtiAmb-
>getAttributeHandle(ms_OrientationStr,
                                            ms_ESClassHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "getAttributeHandle" << endl;
            cerr << &e << endl;
        }
        try{
            Admin::ms_VelocityAttrHandle = Admin::ms_rtiAmb-
>getAttributeHandle(ms_VelocityStr,
                                            ms_ESClassHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "getAttributeHandle" << endl;
            cerr << &e << endl;
        }
```

```cpp
        try{
            Admin::ms_AmmunitionAttrHandle = Admin::ms_rtiAmb-
>getAttributeHandle(ms_AmmunitionStr,
                                            ms_ESClassHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "getAttributeHandle" << endl;
            cerr << &e << endl;
        }
        try{
            Admin::ms_DamageAttrHandle = Admin::ms_rtiAmb-
>getAttributeHandle(ms_DamageStr,
                                            ms_ESClassHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "getAttributeHandle" << endl;
            cerr << &e << endl;
        }


        // RTTI for Interactions
        try{
            Admin::ms_CollisionInteractionHandle =
                Admin::ms_rtiAmb-
>getInteractionClassHandle(ms_CollisionInteractionStr);
        }
        catch (RTI::Exception& e) {
            cerr << "getInteractionClassHandle" << endl;
            cerr << &e << endl;
        }
        try{
            Admin::ms_EntityIdHandle =
                Admin::ms_rtiAmb->getParameterHandle(ms_EntityIdStr,
                    ms_CollisionInteractionHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "getParameterHandle" << endl;
            cerr << &e << endl;
        }

    }// end if
}// end Init

// ************************************************************
// Function Name: PublishAndSubscribe()
// Task:  informs the RTi that this federate will publish and subscribe
//    Entity State Objects and Collision Interactions
// Return Value: void
// ************************************************************

void Admin::PublishAndSubscribe(){

    if ( Admin::ms_rtiAmb ){

        //-------------------------------------------------------
        // To actually subscribe and publish we need to build
```

```cpp
        // an AttributeHandleSet that contains a list of
        // attribute type ids (AttributeHandle).
        //-------------------------------------------------------

        RTI::AttributeHandleSet *ESClassAttributes;
        try {
            ESClassAttributes = RTI::AttributeHandleSetFactory::create(5);
        }
        catch (RTI::Exception& e){
            cerr << "RTI::AttributeHandleSetFactory::create" << endl;
            cerr << &e << endl;
        }
        try {
            ESClassAttributes->add(ms_PositionAttrHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "add" << endl;
            cerr << &e << endl;
        }
        try {
            ESClassAttributes->add(ms_OrientationAttrHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "add" << endl;
            cerr << &e << endl;
        }

        try {
            ESClassAttributes->add(ms_VelocityAttrHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "add" << endl;
            cerr << &e << endl;
        }

        try {
            ESClassAttributes->add(ms_AmmunitionAttrHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "add" << endl;
            cerr << &e << endl;
        }
        try {
            ESClassAttributes->add(ms_DamageAttrHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "add" << endl;
            cerr << &e << endl;
        }

        try {
            Admin::ms_rtiAmb->subscribeObjectClassAttribute( ms_ESClassHandle,
                                            *ESClassAttributes );
        }
        catch (RTI::Exception& e) {
            cerr << "subscribeObjectClassAttribute" << endl;
```

```cpp
            cerr << &e << endl;
        }

        try {
            Admin::ms_rtiAmb->publishObjectClass(
ms_ESClassHandle,*ESClassAttributes);
        }
        catch (RTI::Exception& e) {
            cerr << "publishObjectClass" << endl;
            cerr << &e << endl;
        }

        try {
            Admin::ms_rtiAmb-
>subscribeInteractionClass(ms_CollisionInteractionHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "subscribeObjectClassAttribute" << endl;
            cerr << &e << endl;
        }

        try {
            Admin::ms_rtiAmb->publishInteractionClass(
ms_CollisionInteractionHandle);
        }
        catch (RTI::Exception& e) {
            cerr << "publishInteractionClass" << endl;
            cerr << &e << endl;
        }


        ESClassAttributes->empty();
        delete ESClassAttributes;   // Deallocate the memory
    }//end if

}//end publish and subscribe


// ************************************************************
// Function Name: createFederationExecution()
// Task:  makes RTI call that creates the federation execution if one
//    does already exist
// Return Value: void
// ************************************************************

void Admin::createFederationExecution(){

    cout << "createFederationExecution" << endl;

    try{
        //-------------------------------------------------------
        // A successful createFederationExecution will cause
        // the fedex process to be executed on this machine.
        //-------------------------------------------------------
        Admin::ms_rtiAmb->createFederationExecution( m_FedExecName );
        Sleep(2000);
    }
```

49

```
    catch ( RTI::FederationExecutionAlreadyExists& e ) {
      cerr << "createFederationExecution" << endl;
      cerr << &e << endl;
      Sleep(500);
    }
    catch ( RTI::Exception& e ) {
      cerr << "createFederationExecution" << endl;
      cerr << &e << endl;
    }

}// end createFederationExecution


// ******************************************************
// Function Name:  joinFederation()
// Task:  makes RTI call that joins the federation that has been
//     created
// Return Value: void
// ******************************************************

void Admin::joinFederation(){

    // give join ten tries to get joined
    for (int ix=0;ix<10;ix++){
        try {
            m_federateID = Admin::ms_rtiAmb->joinFederationExecution(
                "Sim_Admin", m_FedExecName, Admin::ms_fedAmb);
            break;
        }
        catch (RTI::FederationExecutionDoesNotExist& e) {
            // no reason to get excited fedex may not be done

            cerr << "Fed Ex does not exist trying again" << endl;
            Sleep(1000);
            continue;
        }
        catch ( RTI::Exception& e ) {
            cerr << "Fatal Error - Join Failed" << endl;
            cerr << &e << endl;
            exit(0);
        }
    }// end for

    //Sleep to ensure federate is joined

    Sleep(3000);


    setTimeManagement();

    //--------------------------------------------------------
    // The Admin class needs to determine what the RTI is
    // going to call its class type and its attribute's types.
    //--------------------------------------------------------
    Admin::Init();
```

```
    PublishAndSubscribe();


}// end join federation

// ******************************************************
// Function Name:  setTimeManagement()
// Task:  sets the RTI's time management state
//     This federate is NOT time managed
// Return Value: void
// ******************************************************

void Admin::setTimeManagement() {
    try {
        ms_rtiAmb->setTimeConstrained( RTI::RTI_FALSE );
        ms_rtiAmb->turnRegulationOff();
    }
    catch ( RTI::Exception& e ){
        cerr << "setTimeManagement" << endl;
        cerr << &e << endl;
    }
}//end setTimeManagement


// ******************************************************
// Function Name:  resignFederate()
// Task:  deletes federate from the fedex and destroys the federation
//     if it is the last federation to resign
// Return Value: void
// ******************************************************

void Admin::resignFederate(){

    cout << "Resigning from federation" << endl;
    try {
        Admin::ms_rtiAmb->resignFederationExecution(
            RTI::DELETE_OBJECTS_AND_RELEASE_ATTRIBUTES);

    }
    catch (RTI::Exception& e){
        cerr << "resignFederationExecution" << endl;
        cerr << &e << endl;

    }

    try {
        Admin::ms_rtiAmb->destroyFederationExecution(m_FedExecName);
    }
    catch (RTI::Exception& e)
    {
        cerr << "destroyFederationExecution" << endl;
        cerr << &e << endl;
    }
}//end resignFederate
```

```
// ******************************************************
// Function Name:  advanceTime(RTI::FederationTime ttime)
// Task:  asks RTI if it is alright to advance time another step
// Return Value: void
// ******************************************************

void Admin::advanceTime(RTI::FederationTime ttime){

    try {
        Admin::ms_rtiAmb->timeAdvanceRequest(ttime);

    }//try

    catch (RTI::Exception& e) {
        cerr << "timeAdvanceRequest" << endl;
        cerr << &e << endl;
    }//catch

}// end advanceTime


// ******************************************************
// Function Name:  tickRTI()
// Task:  ticks the RTi providing processing time to clear buffers
// Return Value: Boolean
// ******************************************************

RTI::Boolean Admin::tickRTI()
{
    RTI::Boolean events;

    try {
        events = Admin::ms_rtiAmb->tick();
    }
    catch(RTI::Exception &e)
    {
        cerr << "tick" << endl;
        cerr << &e << endl;
        return RTI::RTI_FALSE;
    }

    return events;
}


// ******************************************************
// Function Name:  tickRTI(RTI::TickTime minTick,RTI::TickTime maxTick)
// Task:  same as above except ticks RTI for a certain span of time
// Return Value: void
// ******************************************************

RTI::Boolean Admin::tickRTI(RTI::TickTime minTick,RTI::TickTime maxTick)
{
    RTI::Boolean events = RTI::RTI_FALSE;

    try {
        events = Admin::ms_rtiAmb->tick(minTick,maxTick);
```

```
    }
    catch(RTI::Exception &e)
    {
        cerr << "tick" << endl;
        cerr << &e << endl;
    }

    return events;
}

// ******************************************************
// Function Name:  addModuleFunctionPointer(amObject* amObj)
// Task:  Adds the Object associated with each module to a vector that
//     will be used later to instance objects as needed
// Return Value: void
// ******************************************************

void Admin::addModuleFunctionPointer(amObject* amObj) {

    vector<amObject*>::iterator ptr;

    getMutex()->acquire();

    // looks for amObj already in vector
    ptr = find(ms_ModuleArray.begin(),ms_ModuleArray.end(),amObj);

    // if amObj in vector do not add
    if(ptr == ms_ModuleArray.end()){
        ms_ModuleArray.push_back(amObj);
    }
    Admin::ms_NumberModules++;

    getMutex()->release();

} // end addModuleFunctionPointer


// ******************************************************
// Function Name:  findModule(const char* name)
// Task:  returns pointer to the object stored in the object array that
//     corresponds to a particular module
// Return Value: amObject*
// ******************************************************

amObject* Admin::findModule(const char* name){

    amObject* tmp = NULL;
    amObject* retObj = NULL;
    vector<amObject*>::const_iterator   ptr;

    getMutex()->acquire();

    for (ptr = ms_ModuleArray.begin(); ptr != ms_ModuleArray.end(); ptr++){
        tmp = (amObject*)(*ptr);
        if(tmp && strcmp(name,tmp->getModuleName()) == 0){
            retObj = tmp;
```

```
            break;
        }//end if
    }//end for

    getMutex()->release();

    return retObj;

}// end findModule


// ****************************************************************
// Function Name:  moduleLoaded(const char* name)
// Task:  boolean function returning true if module name is loaded
//     false otherwise
// Return Value: Boolean
// ****************************************************************

RTI::Boolean Admin::moduleLoaded(const char* name){

    RTI::Boolean flag = RTI::RTI_FALSE;
    bbModule* mod = bbModule::findObject(name);

    if (mod) {
        flag = RTI::RTI_TRUE;
    }// end if

    return flag;
}// end moduleLoaded


// ****************************************************************
// Function Name:  addSimEntity(const char* name, RTI::ObjectID oid)
// Task:  adds simulation entity to the amObjectArray
// Return Value: amObject*
// ****************************************************************

amObject* Admin::addSimEntity(const char* name, RTI::ObjectID oid){

    amObject* mod = Admin::findModule(name);
    amObject* tmp = NULL;
    if(mod && mod->isTerrain()){
        Admin::ms_Terrain = mod->createObject(oid);
    }//end if
    else if (mod){
        tmp = mod->createObject(oid);
        vector<amObject*>::iterator ptr;

        getMutex()->acquire();

        ms_amObjectArray.push_back(tmp);
        Admin::ms_NumberAmObjects++;

        getMutex()->release();

    }//end if
    return tmp;
```

```
}// end addSimEntity


// ****************************************************************
// Function Name:  loadModule(const char* name)
// Task:  makes bamboo call that loads the name module
// Return Value: void
// ****************************************************************

void Admin::loadModule(const char* name){
    bbModule* mod;

    getMutex()->acquire();

    mod = bbModule::load(name,ms_ModuleURL);

    getMutex()->release();

}// end loadModule


// ****************************************************************
// Function Name:  unloadModule(const char* name)
// Task:  makes bamboo call that unloads the name module
// Return Value: void
// ****************************************************************

void Admin::unloadModule(const char* name){

    bbModule* mod;
    mod = bbModule::findObject(name);

    getMutex()->acquire();

    bbModule::unload(mod);

    getMutex()->release();

} // end unloadModule


// ****************************************************************
// Function Name:  updateFederation(RTI::FederationTime& newTime)
// Task:  iterates the amObjectArray telling each localObject to update
//     the federation with their current state
// Return Value: void
// ****************************************************************

void Admin::updateFederation(RTI::FederationTime& newTime){
    amObject* tmp = NULL;


    vector<amObject*>::iterator    ptr;

    getMutex()->acquire();
```

```
    // iterate the amObjectArray
    for (ptr = ms_amObjectArray.begin(); ptr != ms_amObjectArray.end();
ptr++){
        tmp = (amObject*)(*ptr);
        if(tmp){
            if(tmp->localObject()){
                tmp->sendUpdates(newTime);
            }//end if;
        }//end if
    }//end for

    // do same for the terrain
    if(ms_Terrain){
        if(ms_Terrain->localObject()){
            getMutex()->acquire();
            ms_Terrain->sendUpdates(newTime);
            getMutex()->release();
        }
    }//end if

    getMutex()->release();

}//end updateFederation


// ****************************************************************
// Function Name:  display()
// Task:  iterates the amObjectArray telling each amObject to display
//     its geometry in the proper position and orientation
// Return Value: void
// ****************************************************************

void Admin::display(){
    amObject* tmp = NULL;

    if(ms_Terrain){
        ms_Terrain->display();
    }//end if

    vector<amObject*>::iterator ptr;

    getMutex()->acquire();

    ptr = Admin::ms_amObjectArray.begin();
    while(ptr != Admin::ms_amObjectArray.end()){
        tmp = (amObject*)*ptr;
        if(tmp)
            tmp->display();
        ptr++;
    }//end while

    getMutex()->release();

}// end display


// ****************************************************************
```

```
// Function Name:  receiveUpdate( RTI::ObjectID theObject,
//         const RTI::AttributeHandleValuePairSet& theAttributes,
//         RTI::FederationTime              theTime,
//         const RTI::UserSuppliedTag       theTag,
//         RTI::EventRetractionHandle        theHandle )
// Task:  sends the handle value pair to the correct amObject.
//     if the moduel is not loaded then the proper calls are made to
//     load the module for the object to be updated
// Return Value: void
// ****************************************************************

void Admin::receiveUpdate( RTI::ObjectID theObject,
        const RTI::AttributeHandleValuePairSet& theAttributes,
        RTI::FederationTime              theTime,
        const RTI::UserSuppliedTag       theTag,
        RTI::EventRetractionHandle        theHandle ){

    // find the correct object
    amObject* tmp = Admin::findSimEntity(theObject);

    if (tmp){
        getMutex()->acquire();

        tmp->receiveUpdates(theObject,theAttributes,theTime,theTag);

        getMutex()->release();
    }// end if

    // add module if necessay or add simEntity
    else {

        cout << "Object " << theObject << " not found - adding" << endl;
        // is module loaded
        if (Admin::moduleLoaded(theTag)){
            cout << "adding new " << theTag << endl;
            Admin::addSimEntity(theTag,theObject);
        }// end if

        // if module not loaded
        else {
            cout << "Loading Module " << theTag << endl;
            Admin::loadModule(theTag);
            Admin::addSimEntity(theTag, theObject);
        }// end else

    }// end else

}//end receiveUpdate


// ****************************************************************
// Function Name: receiveInteraction( RTI::InteractionClassHandle
//     theInteraction,
//         const RTI::ParameterHandleValuePairSet& theParameters,
//         RTI::FederationTime theTime,
//         const RTI::UserSuppliedTag theTag,
//         RTI::EventRetractionHandle theHandle
```

```cpp
// Task:  sends interaction data to proper simEntity
// Return Value: void
// *************************************************************

void Admin::receiveInteraction( RTI::InteractionClassHandle theInteraction,
          const RTI::ParameterHandleValuePairSet& theParameters,
          RTI::FederationTime theTime,
          const RTI::UserSuppliedTag theTag,
          RTI::EventRetractionHandle theHandle){


    if(Admin::ms_InteractionFlag){

        RTI::ObjectID oid = 0;
        if(theInteraction == ms_CollisionInteractionHandle){
            RTI::AttributeHandle paramHandle;
            unsigned long      valueLength;

            for ( unsigned int ix = 0; ix < theParameters.size(); ix++ ){
                paramHandle = theParameters.getHandle( ix );
                if ( paramHandle == ms_EntityIdHandle ){
                    theParameters.getValue( ix, (char*)&oid, valueLength );
                }//end if
            }//end for

            // find the correct object

            amObject* tmp = Admin::findSimEntity((RTI::ObjectID)oid);
            if(tmp){
                getMutex()->acquire();

                tmp->receiveInteraction(theInteraction,theParameters);

                getMutex()->release();
            }//end if
        }//end if
        else {
            cout << "Interaction not Known " << theInteraction << endl;
        }//end else
    }//end if InteractionFlag

}// end receiveInteraction


// *************************************************************
// Function Name:  sendCollisionInteraction(RTI::FederationTime
//    theTime, RTI::ObjectID oid)
// Task:  sends Collisioninteraction data to federation
// Return Value: void
// *************************************************************

void Admin::sendCollisionInteraction(RTI::FederationTime theTime,
                                     RTI::ObjectID oid){

    RTI::ParameterHandleValuePairSet* params = NULL;
    RTI::ULong numparams(1);
    params = RTI::ParameterSetFactory::create(numparams);
```

```cpp
    params->add(getEntityIdHandle(), (char*) oid, sizeof(RTI::ObjectID));

    getMutex()->acquire();

    try{

        Admin::ms_rtiAmb->sendInteraction(getCollisionInteractionHandle(),
            *params,theTime,NULL);
    }
    catch(RTI::Exception& e){
        cerr << &e << endl;
    }
    getMutex()->release();
}// end sendCollisionInteraction


// *************************************************************
// Function Name:  sendEntityUpdate(RTI::ObjectID theObject,
//        RTI::AttributeHandleValuePairSet& theAttributes,
//        const RTI::UserSuppliedTag theTag)
// Task:  sends the handle value pair from an entity to the federation
// Return Value: void
// *************************************************************

void Admin::sendEntityUpdate(RTI::ObjectID theObject,
        RTI::AttributeHandleValuePairSet& theAttributes,
        const RTI::UserSuppliedTag theTag){

    RTI::FederationTime  theTime = m_lastTime + ms_grantTime;

    if ( Admin::ms_rtiAmb ){
        //---------------------------------------------------------
        // Update state of Boid
        //---------------------------------------------------------
        getMutex()->acquire();

        try
        {
            Admin::ms_rtiAmb->updateAttributeValues( theObject,
                    theAttributes, theTime, theTag );
            // Must free the memory
            theAttributes.empty();
        }//end try
        catch ( RTI::Exception& e )
        {
            cerr << &e << endl;
        }//end catch

        getMutex()->release();

    }//end if

    m_lastTime = theTime;

}//end sendEntityUpdate
```

```cpp
// *************************************************************
// Function Name:  removeAmObject(RTI::ObjectID oid)
// Task:  does house keeping required to remove a simEntity from the
//    federation
// Return Value: void
// *************************************************************

void Admin::removeAmObject(RTI::ObjectID oid){

    amObject* pBoid = NULL;
    unsigned int ndx = 0;

    // check if item to be deleted is the terrain module
    if ( Admin::ms_Terrain && Admin::ms_Terrain->getEntityID() == oid ){

        if(Admin::ms_Terrain->localObject()){

            getMutex()->acquire();

            try{
                Admin::ms_rtiAmb->deleteObject( Admin::ms_Terrain->getEntityID(),
                    0.0, NULL );
            }//end try
            catch(RTI::Exception &e){
                cerr << &e << endl;
            }//end catch

            getMutex()->release();

        }//end if
        Admin::ms_Terrain->deleteObject();
        Admin::ms_Terrain = NULL;

    }

    else {  // not a Terrain entity

        //---------------------------------------------------------
        // Find the instance.
        //---------------------------------------------------------
        vector<amObject*>::iterator    ptr;
        amObject* tmp = NULL;

        getMutex()->acquire();

        for (ptr = ms_amObjectArray.begin(); ptr != ms_amObjectArray.end();
ptr++){
            tmp = (amObject*)(*ptr);
            if(tmp && (tmp->getEntityID() == oid)){
                pBoid = tmp;
                break;
            }//end if
        }//end for
```

```cpp
        if ( pBoid ){
            Admin::ms_NumberAmObjects--;

            //---------------------------------------------------------
            // If the instance is a local object
            // we should delete it from the RTI space.
            //---------------------------------------------------------
            if ( Admin::ms_rtiAmb && pBoid->localObject() ){
                try{
                    Admin::ms_rtiAmb->deleteObject( pBoid->getEntityID(),
                        0.0, NULL );
                }
                catch(RTI::Exception &e){
                    cerr << &e << endl;
                }
            }// end if
            else {
                //---------------------------------------------------------
                // Otherwise, this is a remote object that removeObject was
                // called on.
                //---------------------------------------------------------
                //We don't need to do anything here
            }//end else
            ms_amObjectArray.erase(ptr);
            pBoid->deleteObject();

        }//end if

        getMutex()->release();

    }// end else
}//end removeAmObject


// *************************************************************
// Function Name:  removeModule(const char* moduleName)
// Task:  removes module object form model array
// Return Value: void
// *************************************************************

void Admin::removeModule(const char* moduleName){
    amObject* tmpObject = NULL;
    amObject* tmp = NULL;
    unsigned int ndx = 0;
    //---------------------------------------------------------
    // Find the position of this instance.
    //---------------------------------------------------------
    vector<amObject*>::iterator    ptr;

    getMutex()->acquire();

    for (ptr = ms_ModuleArray.begin(); ptr != ms_ModuleArray.end(); ptr++){
        tmp = (amObject*)(*ptr);
        if(tmp && (strcmp(tmp->getModuleName(),moduleName)==0)){
            tmpObject = tmp;
            ms_ModuleArray.erase(ptr);
            break;
```

```
            }//end if
        }//end for

        getMutex()->release();

        if ( tmpObject )
        {

            Admin::ms_NumberModules--;

            delete tmpObject;

        }//endif

}//end remove Module


// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name:  checkEntityCollision(amObject* that)
// Task:  iterates ObjectArray looking for collisions with other
//     entities
// Return Value: amObject* it collided with
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

amObject* Admin::checkEntityCollision(amObject* that){

    amObject* tmp = NULL;
    amObject* retObj = NULL;

    vector<amObject*>::iterator ptr;
    ptr = Admin::ms_amObjectArray.begin();

    getMutex()->acquire();

    while(ptr != Admin::ms_amObjectArray.end()){
        tmp = (amObject*)*ptr;
        if (tmp != that){
            if(tmp->checkCollision(that)){
                retObj = tmp;
                break;
            }//end if
        }//end if
        ptr++;
    }//end while

    getMutex()->release();

    return retObj;

}//end checkentityCollision


// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name:  getMutex()
// Task:  pointer to Admin class Mutex
// Return Value: ACE_Recursive_Thread_Mutex*
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
```

```
ACE_Recursive_Thread_Mutex* Admin::getMutex(){
    return &adminMutex;
}


// end admin .c
```

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
//     EXECUTIVE SUMMARY
//
// File Name: AdminFederateAmbassador.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Method Definitions of AdminFederateAmbassador class
//     This file defines Federate ambassador functions required for
//     proper execution in this federation
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

#include "Rti.hh"
#ifndef _AdminFederateAmbassador
#define _AdminFederateAmbassador


class AdminFederateAmbassador : public RTI::FederateAmbassador
{
public:
/////////////////////////////////////////
// Federation Management Services //
/////////////////////////////////////////

// 2.6
virtual void initiatePause (
  const RTI::PauseLabel label) // supplied C4
throw (
  RTI::FederateAlreadyPaused,
  RTI::FederateInternalError);

// 2.9
virtual void initiateResume ()
throw (
  RTI::FederateNotPaused,
  RTI::FederateInternalError);

// 2.12
virtual void initiateFederateSave (
  const RTI::SaveLabel label) // supplied C4
throw (
  RTI::UnableToPerformSave,
  RTI::FederateInternalError);

virtual void initiateFederateSave (
  const RTI::SaveLabel    label,    // supplied C4
        RTI::FederationTime theTime) // supplied C1
throw (
  RTI::InvalidFederationTime,
  RTI::UnableToPerformSave,
  RTI::FederateInternalError);

// 2.16
virtual void initiateRestore (
  const RTI::SaveLabel label) // supplied C4
```

```
throw (
  RTI::SpecifiedSaveLabelDoesNotExist,
  RTI::CouldNotRestore,
  RTI::FederateInternalError);

/////////////////////////////////////////
// Declaration Management Services //
/////////////////////////////////////////

// 3.5
virtual void startUpdates (
        RTI::ObjectClassHandle    theClass,        // supplied C1
  const RTI::AttributeHandleSet& theAttributes) // supplied C4
throw (
  RTI::ObjectClassNotPublished,
  RTI::AttributeNotPublished,
  RTI::FederateInternalError);

virtual void stopUpdates (
        RTI::ObjectClassHandle    theClass,        // supplied C1
  const RTI::AttributeHandleSet& theAttributes) // supplied C4
throw (
  RTI::ObjectClassNotPublished,
  RTI::AttributeNotPublished,
  RTI::FederateInternalError);

// 3.6
virtual void startInteractionGeneration (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError);

virtual void stopInteractionGeneration (
  RTI::InteractionClassHandle theHandle) // supplied C1
throw (
  RTI::InteractionClassNotPublished,
  RTI::FederateInternalError);

/////////////////////////////////////////
// Object Management Services //
/////////////////////////////////////////

// 4.4
virtual void discoverObject (
        RTI::ObjectID            theObject,      // supplied C1
        RTI::ObjectClassHandle   theObjectClass, // supplied C1
        RTI::FederationTime      theTime,        // supplied C1
  const RTI::UserSuppliedTag     theTag,         // supplied C4
        RTI::EventRetractionHandle theHandle)      // supplied C1
throw (
  RTI::CouldNotDiscover,
  RTI::ObjectClassNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError);

// 4.5
```

```
virtual void reflectAttributeValues (
        RTI::ObjectID                      theObject,      // supplied C1
  const RTI::AttributeHandleValuePairSet&  theAttributes,  // supplied C4
        RTI::FederationTime                theTime,        // supplied C1
  const RTI::UserSuppliedTag               theTag,         // supplied C4
        RTI::EventRetractionHandle         theHandle)      // supplied C1
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError);

// 4.7
virtual void receiveInteraction (
        RTI::InteractionClassHandle        theInteraction, // supplied C1
  const RTI::ParameterHandleValuePairSet&  theParameters,  // supplied C4
        RTI::FederationTime                theTime,        // supplied C1
  const RTI::UserSuppliedTag               theTag,         // supplied C4
        RTI::EventRetractionHandle         theHandle)      // supplied C1
throw (
  RTI::InteractionClassNotKnown,
  RTI::InteractionParameterNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError);

// 4.9
virtual void removeObject (
        RTI::ObjectID              theObject, // supplied C1
        RTI::ObjectRemovalReason   theReason, // supplied C1
        RTI::FederationTime        theTime,   // supplied C1
  const RTI::UserSuppliedTag       theTag,    // supplied C4
        RTI::EventRetractionHandle theHandle) // supplied C1
throw (
  RTI::ObjectNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError);

virtual void removeObject (
  RTI::ObjectID            theObject, // supplied C1
  RTI::ObjectRemovalReason theReason) // supplied C1
throw (
  RTI::ObjectNotKnown,
  RTI::InvalidFederationTime,
  RTI::FederateInternalError);

// 4.15
virtual void provideAttributeValueUpdate (
        RTI::ObjectID            theObject,     // supplied C1
  const RTI::AttributeHandleSet& theAttributes) // supplied C4
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeNotKnown,
  RTI::FederateInternalError);

// 4.17
virtual void reflectRetraction (
  RTI::EventRetractionHandle theHandle) // supplied C1
```

```
throw (
  RTI::EventNotKnown,
  RTI::FederateInternalError);

///////////////////////////////////////
// Ownership Management Services //
///////////////////////////////////////

// 5.2
virtual RTI::AttributeHandleSet&                          // returned C6
requestAttributeOwnershipAssumption (
        RTI::ObjectID            theObject,         // supplied C1
  const RTI::AttributeHandleSet& offeredAttributes, // supplied C4
  const RTI::UserSuppliedTag     theTag)            // supplied C4
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeAlreadyOwned,
  RTI::FederateInternalError);

virtual RTI::AttributeHandleSet&                          // returned C6
requestAttributeOwnershipAssumption (
        RTI::ObjectID            theObject,          // supplied C1
  const RTI::AttributeHandleSet& offeredAttributes) // supplied C4
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeAlreadyOwned,
  RTI::FederateInternalError);

// 5.3
virtual void attributeOwnershipDivestitureNotification (
        RTI::ObjectID            theObject,          // supplied C1
  const RTI::AttributeHandleSet& releasedAttributes) // supplied C4
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeNotKnown,
  RTI::FederateInternalError);

// 5.4
virtual void attributeOwnershipAcquisitionNotification (
        RTI::ObjectID            theObject,          // supplied C1
  const RTI::AttributeHandleSet& securedAttributes) // supplied C4
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeNotKnown,
  RTI::FederateInternalError);

// 5.6
virtual RTI::AttributeHandleSet&                          // returned C6
requestAttributeOwnershipRelease (
        RTI::ObjectID            theObject,            // supplied C1
  const RTI::AttributeHandleSet& candidateAttributes, // supplied C4
  const RTI::UserSuppliedTag     theTag)              // supplied C4
throw (
  RTI::ObjectNotKnown,
  RTI::AttributeNotKnown,
  RTI::FederateInternalError);
```

```
// 5.7
virtual void informAttributeOwnership (
  RTI::ObjectID         theObject,    // supplied C1
  RTI::AttributeHandle  theAttribute, // supplied C1
  RTI::FederateHandle   theOwner)     // supplied C1
throw (
  RTI::FederateInternalError);

///////////////////////////////////////
// Time Management Services //
///////////////////////////////////////

// 6.10
virtual void timeAdvanceGrant (
  RTI::FederationTime theTime) // supplied C1
throw (
  RTI::InvalidFederationTime,
  RTI::TimeAdvanceWasNotInProgress,
  RTI::FederationTimeAlreadyPassed,
  RTI::FederateInternalError);

///////////////////////////////////////
// Data Distribution Management //
///////////////////////////////////////

// 7.4
virtual void changeThresholds (
  RTI::Region        theRegion,     // supplied
  RTI::ThresholdSet& theThresholds) // returned
throw (
  RTI::RegionNotKnown,
  RTI::FederateInternalError);
// 6:17 PM 8/28/96    federateAmbServices.hh  5

protected:

private:

};
#endif
```

```
// ****************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: AdminFederateAmbassador.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Method Definitions of AdminFederateAmbassador class
//    This file defines Federate ambassador functions required for
//    proper execution in this federation
//
// September 1998, Masters Thesis
// ****************************************************************

#include "AdminFederateAmbassador.h"
#include "Rti.hh"
#include "admin.h"

/////////////////////////////////////////////////////////////////////
// AdminFederateAmbassador Class
/////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// Construction/Destruction
/////////////////////////////////////////////////////////////////////


///////////////////////////////////////
// Federation Management Services //
///////////////////////////////////////

void AdminFederateAmbassador::initiatePause( const RTI::PauseLabel label )
        throw (RTI::FederateAlreadyPaused,
               RTI::FederateInternalError)
{
    // Not implemented in 1.0
}


void AdminFederateAmbassador::initiateResume()
        throw (RTI::FederateNotPaused,
               RTI::FederateInternalError)
{
    // Not implemented in 1.0
}

void AdminFederateAmbassador::initiateFederateSave( const RTI::SaveLabel
label )
        throw (RTI::UnableToPerformSave,
               RTI::FederateInternalError)
{
    // Not implemented in 1.0
#ifdef TEST_SAVE
    saveFlag = RTI::RTI_TRUE;
    cerr << "initiateFederateSave called with label = " << label << endl;
#endif
}
```

```
void AdminFederateAmbassador::initiateFederateSave(const RTI::SaveLabel
label,
                                 RTI::FederationTime theTime)
    throw (RTI::InvalidFederationTime,
           RTI::UnableToPerformSave,
           RTI::FederateInternalError)
{
    // Not implemented in 1.0
}

void AdminFederateAmbassador::initiateRestore(const RTI::SaveLabel label)
     throw (RTI::SpecifiedSaveLabelDoesNotExist,
            RTI::CouldNotRestore,
            RTI::FederateInternalError)
{
    // Not implemented in 1.0
}

///////////////////////////////////////
// Declaration Management Services //
///////////////////////////////////////
void AdminFederateAmbassador::startUpdates(RTI::ObjectClassHandle   theClass,
                                 const RTI::AttributeHandleSet&
theAttributes)
     throw (RTI::ObjectClassNotPublished,
            RTI::AttributeNotPublished,
            RTI::FederateInternalError)
{
cerr << "AFA::startUpdates" << endl;

    //-------------------------------------------------------------------
    // Gets called immediately in 1.0 for all classes you publish.
    //-------------------------------------------------------------------
}

void AdminFederateAmbassador::stopUpdates(RTI::ObjectClassHandle  theClass,
                                 const RTI::AttributeHandleSet&
theAttributes)
     throw (RTI::ObjectClassNotPublished,
            RTI::AttributeNotPublished,
            RTI::FederateInternalError)
{
cerr << "stopUpdates" << endl;
    //-------------------------------------------------------------------
    // Never gets called in 1.0 but we will implement it for good form.
    //-------------------------------------------------------------------

}

void AdminFederateAmbassador::startInteractionGeneration
      (RTI::InteractionClassHandle theClass)
      throw (RTI::InteractionClassNotPublished,
             RTI::FederateInternalError)
{
cerr << "startInteractionGeneration" << endl;
```

```
Admin::ms_InteractionFlag = RTI::RTI_TRUE;
}

void AdminFederateAmbassador::stopInteractionGeneration
      (RTI::InteractionClassHandle theClass)
      throw (RTI::InteractionClassNotPublished,
             RTI::FederateInternalError)
{
cerr << "stopINteractionGeneration" << endl;

Admin::ms_InteractionFlag = RTI::RTI_FALSE;

}

///////////////////////////////////////
// Object Management Services //
///////////////////////////////////////
void AdminFederateAmbassador::discoverObject( RTI::ObjectID
theObject,
                                 RTI::ObjectClassHandle
theObjectClass,
                                 RTI::FederationTime
theTime,
                                 const RTI::UserSuppliedTag theTag,
                                 RTI::EventRetractionHandle
theHandle)
      throw (RTI::CouldNotDiscover,
             RTI::ObjectClassNotKnown,
             RTI::InvalidFederationTime,
             RTI::FederateInternalError)
{
cout << "AFA::discovered object " << theObject << " Module " << theTag <<
endl;

    // is this an object type I know about
    if (theObjectClass == Admin::getESClassHandle() ) {

        // is module loaded
        if (Admin::moduleLoaded(theTag)){
            Admin::addSimEntity(theTag,theObject);
        }// end if

        // if module not loaded
        else {
            Admin::loadModule(theTag);
            Admin::addSimEntity(theTag, theObject);
        }// end else

    }// endif
    else {

        cerr << "Discovered Obect class unknown to me." << endl;

    }// end else
```

```
}// end discover object

void AdminFederateAmbassador::reflectAttributeValues
      ( RTI::ObjectID                        theObject,
        const RTI::AttributeHandleValuePairSet& theAttributes,
        RTI::FederationTime                  theTime,
        const RTI::UserSuppliedTag            theTag,
        RTI::EventRetractionHandle            theHandle )
      throw (RTI::ObjectNotKnown,
             RTI::AttributeNotKnown,
             RTI::InvalidFederationTime,
             RTI::FederateInternalError)
{
    Admin::receiveUpdate(theObject,theAttributes,theTime,theTag,theHandle);

}// end reflectAttributeValues

void AdminFederateAmbassador::receiveInteraction
      ( RTI::InteractionClassHandle theInteraction,
        const RTI::ParameterHandleValuePairSet& theParameters,
        RTI::FederationTime theTime,
        const RTI::UserSuppliedTag theTag,
        RTI::EventRetractionHandle theHandle)
      throw (RTI::InteractionClassNotKnown,
             RTI::InteractionParameterNotKnown,
             RTI::InvalidFederationTime,
             RTI::FederateInternalError)
{

Admin::receiveInteraction(theInteraction,theParameters,theTime,theTag,theHand
le);

}

void AdminFederateAmbassador::removeObject( RTI::ObjectID
theObject,
                                 RTI::ObjectRemovalReason
theReason,
                                 RTI::FederationTime        theTime,
                                 const RTI::UserSuppliedTag theTag,
                                 RTI::EventRetractionHandle theHandle
      )
      throw (RTI::ObjectNotKnown,
             RTI::InvalidFederationTime,
             RTI::FederateInternalError)
{
    //-------------------------------------------------------------------
    // Call the other removeObject method since this should probably
    // be implemented using default parameter values.
    //-------------------------------------------------------------------
    removeObject(theObject,theReason);
}

void AdminFederateAmbassador::removeObject( RTI::ObjectID theObject,
                                 RTI::ObjectRemovalReason   theReason)
      throw (RTI::ObjectNotKnown,
```

```
             RTI::InvalidFederationTime,
             RTI::FederateInternalError)
{
cout << "AFA::removeObject" << endl;

    Admin::removeAmObject(theObject);

}

void AdminFederateAmbassador::provideAttributeValueUpdate
      ( RTI::ObjectID                theObject,
        const RTI::AttributeHandleSet& theAttributes)
      throw (RTI::ObjectNotKnown,
             RTI::AttributeNotKnown,
             RTI::FederateInternalError)
{

}

void AdminFederateAmbassador::reflectRetraction( RTI::EventRetractionHandle
theHandle )
      throw (RTI::EventNotKnown,
             RTI::FederateInternalError)
{
    cerr << "AdminFederateAmbassador::reflectRetraction: handle = " << endl;
    // I didn't implement this one - sorry!
}

///////////////////////////////////////
// Ownership Management Services //
///////////////////////////////////////

RTI::AttributeHandleSet&
AdminFederateAmbassador::requestAttributeOwnershipAssumption
      ( RTI::ObjectID                theObject,
        const RTI::AttributeHandleSet& offeredAttributes,
        const RTI::UserSuppliedTag     theTag)
        throw (RTI::ObjectNotKnown,
               RTI::AttributeAlreadyOwned,
               RTI::FederateInternalError)
{
    // I didn't implement this one - sorry!

    // The following is to allow it to compile with the SparcWorks compiler
    RTI::AttributeHandleSet* dummy =
RTI::AttributeHandleSetFactory::create(0);
    return *dummy;
}

RTI::AttributeHandleSet&
AdminFederateAmbassador::requestAttributeOwnershipAssumption
      ( RTI::ObjectID                theObject,
        const RTI::AttributeHandleSet& offeredAttributes )

        throw (RTI::ObjectNotKnown,
               RTI::AttributeAlreadyOwned,
               RTI::FederateInternalError)
```

55

```
{
    // I didn't implement this one - sorry!

    // The following is to allow it to compile with the SparcWorks compiler
    RTI::AttributeHandleSet* dummy =
RTI::AttributeHandleSetFactory::create(0);
    return *dummy;
}

void AdminFederateAmbassador::attributeOwnershipDivestitureNotification
        ( RTI::ObjectID              theObject,
          const RTI::AttributeHandleSet& releasedAttributes)
    throw (RTI::ObjectNotKnown,
           RTI::AttributeNotKnown,
           RTI::FederateInternalError)
{
    // I didn't implement this one - sorry!
}

void AdminFederateAmbassador::attributeOwnershipAcquisitionNotification
        ( RTI::ObjectID              theObject,
          const RTI::AttributeHandleSet& securedAttributes )
    throw (RTI::ObjectNotKnown,
           RTI::AttributeNotKnown,
           RTI::FederateInternalError)
{
    // I didn't implement this one - sorry!
}

RTI::AttributeHandleSet&
AdminFederateAmbassador::requestAttributeOwnershipRelease
    ( RTI::ObjectID theObject,
      const RTI::AttributeHandleSet& candidateAttributes,
      const RTI::UserSuppliedTag theTag )
    throw (RTI::ObjectNotKnown,
           RTI::AttributeNotKnown,
           RTI::FederateInternalError)
{
    // I didn't implement this one - sorry!

    // The following is to allow it to compile with the SparcWorks compiler
    RTI::AttributeHandleSet* dummy =
RTI::AttributeHandleSetFactory::create(0);
    return *dummy;
}

void AdminFederateAmbassador::informAttributeOwnership
        ( RTI::ObjectID           theObject,
          RTI::AttributeHandle theAttribute,
          RTI::FederateHandle  theOwner)
    throw (RTI::FederateInternalError)
{
    // I didn't implement this one - sorry!
}

///////////////////////////////////
// Time Management Services //
```

```
///////////////////////////////////

void AdminFederateAmbassador::timeAdvanceGrant( RTI::FederationTime theTime )
    throw (RTI::InvalidFederationTime,
           RTI::TimeAdvanceWasNotInProgress,
           RTI::FederationTimeAlreadyPassed,
           RTI::FederateInternalError)
{

    //---------------------------------------------------------------
    // Set the time granted to and the flag to let me know
    // time has changed.
    //---------------------------------------------------------------

//   Admin::ms_grantTime   = theTime;
//   Admin::ms_timeAdvGrant = RTI::RTI_TRUE;

}

///////////////////////////////////////////////////////////////////
// Data Distribution Management  - NOT IMPLEMENTED IN 1.0//
///////////////////////////////////////////////////////////////////

void AdminFederateAmbassador::changeThresholds( RTI::Region        theRegion,
                                                RTI::ThresholdSet& theThresholds
    )
    throw (RTI::RegionNotKnown,
           RTI::FederateInternalError)
{
    // Not implemented in 1.0
}
```

```
// **********************************************************
//   EXECUTIVE SUMMARY
//
// File Name: amObject.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: pure virtual class defining the interface between
//    adminModule objects, the federate Ambassador and rti ambassador
//
// September 1998, Masters Thesis
// **********************************************************
#include "npsQuaternion.h"
#include "npsVec3f.h"
#include "Rti.hh"
#include "ace/OS.h"

#ifndef _amObject
#define _amObject

#ifndef SIM_API
  #ifdef VISUALCPP
    #define SIM_API __declspec(dllimport)
  #else
    #define SIM_API
  #endif
#endif


class amObject
{
public:
    // member variables
    RTI::ObjectID        entityID;
    npsVec3f             position;
    npsQuaternion        orientation;
    npsVec3f             velocity;
    int                  ammunition;
    float                damage;

    amObject(RTI::ObjectID);
    amObject();

    //getters of Entity State fields

    RTI::ObjectID  getEntityID();
    npsVec3f       getPosition();
    npsQuaternion  getOrientation();
    npsVec3f       getVelocity();
    int            getAmmunition();
    float          getDamage();

    //setters of Entity State fields

    void setEntityID(RTI::ObjectID);
    void setPosition(npsVec3f);
    void setOrientation(npsQuaternion);
```

```
    void setVelocity(npsVec3f);
    void setAmmunition(int);
    void setDamage(float);

    // RTI ambassdor interface
    virtual void sendUpdates(RTI::FederationTime theTime) = 0;

    virtual void sendUpdates(RTI::ObjectID,RTI::AttributeHandleValuePairSet&,
        RTI::FederationTime, RTI::UserSuppliedTag) = 0;

    virtual void sendInteraction(RTI::InteractionClassHandle,
        RTI::ParameterHandleValuePairSet&, RTI::FederationTime,
        RTI::UserSuppliedTag) = 0;

    // FederateAmbassador interface
    virtual void receiveUpdates(RTI::ObjectID oid,
        const RTI::AttributeHandleValuePairSet& ta,
        RTI::FederationTime ft, RTI::UserSuppliedTag tag) = 0;

    virtual void receiveInteraction(RTI::InteractionClassHandle,
        RTI::ParameterHandleValuePairSet&, RTI::FederationTime,
        RTI::UserSuppliedTag) = 0;

    virtual void receiveInteraction(RTI::InteractionClassHandle,
        const RTI::ParameterHandleValuePairSet&) = 0;

    // Object display

    virtual void display() = 0;

    virtual amObject* createObject(RTI::ObjectID) = 0;

    virtual void deleteObject() = 0;

    virtual char* getModuleName() = 0;

    virtual RTI::Boolean localObject() = 0;

    virtual RTI::Boolean checkCollision(amObject* that) = 0;

    virtual float checkTerrainCollision(amObject* that)= 0;

    virtual RTI::Boolean isTerrain() = 0;

}; // end amObject

#endif
```

```
// **************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: amObject.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: Defines the base object that must be inherited by all
//    participating in the federation
//
// September 1998, Masters Thesis
// **************************************************************


#include "amObject.h"


amObject::amObject(RTI::ObjectID oid) {

    entityID = oid;

}// end amObject

amObject::amObject(){}


    //getters of Entity State fields

RTI::ObjectID  amObject::getEntityID(){ return entityID;}
npsVec3f       amObject::getPosition(){return position;}
npsQuaternion  amObject::getOrientation(){return orientation;}
npsVec3f       amObject::getVelocity(){return velocity;}
int            amObject::getAmmunition(){return ammunition;}
float          amObject::getDamage(){return damage;}

//setters of Entity State fields

void amObject::setEntityID(RTI::ObjectID oid){entityID = oid;}
void amObject::setPosition(npsVec3f number){position = number;}
void amObject::setOrientation(npsQuaternion number){orientation = number;}
void amObject::setVelocity(npsVec3f number){velocity = number;}
void amObject::setAmmunition(int number){ammunition = number;}
void amObject::setDamage(float number){damage = number;}
```

amBoid Files

```
This is the module.txt for the amBoid module


Bamboo1.0b
4
npsVisualModule
bbKeyboardModule
npsFlyingCameraModule
amHLAAdmin
amCheckerboard
```

```
// **************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: module.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// **************************************************************

#ifndef __module
#define __module


// **************************************************************
//    INCLUDES AND EXTERNS
// **************************************************************


// **************************************************************
//    DEFINES
// **************************************************************


// **************************************************************
//    FUNCTION PROTOTYPE SPECIFICATIONS
// **************************************************************


#ifdef __cplusplus
extern "C" {
#endif

  ADMIN_API const char *getModuleName();
  ADMIN_API float getModuleVersion();
  ADMIN_API const char *getModuleDate();
  ADMIN_API const char *getModuleText();
  ADMIN_API bool initModule();
  ADMIN_API bool exitModule();

#ifdef __cplusplus
};
#endif


// **************************************************************
//    INLINED MEMBER FUNCTIONS
// **************************************************************


#endif // __module
```

```
// ****************************************************
//    EXECUTIVE SUMMARY
//
// File Name: module.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ****************************************************

// ****************************************************
//    INCLUDES AND EXTERNS
// ****************************************************

#include <stdio.h>
#include <stdlib.h> // atof
#include "module.h"
#include "amBoid.h"

// ****************************************************
//    CODE
// ****************************************************

const char *getModuleName()
  {
  return "amBoid";
  }

float getModuleVersion()
  {
  return 1.0;
  }

const char *getModuleDate()
  {
  return "1998/09/01 06:05:48";
  }


const char *getModuleText()
  {
  return "This is a really nice module";
  }

bool initModule()
  {
  initamBoid();
  return 1;
  }

bool exitModule()
  {
   exitamBoid();
   return 1;
  }
```

```
// ****************************************************
//    EXECUTIVE SUMMARY
//
// File Name: amBoid.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ****************************************************

#ifndef _amBoid
#define _amBoid


// ****************************************************
//    INCLUDES AND EXTERNS
// ****************************************************

// ****************************************************
//    DEFINES
// ****************************************************


// ****************************************************
//    FUNCTION PROTOTYPE SPECIFICATIONS
// ****************************************************

void initamBoid();

void exitamBoid();


// ****************************************************
//    INLINED MEMBER FUNCTIONS
// ****************************************************


#endif //
```

```
// ****************************************************
//    EXECUTIVE SUMMARY
//
// File Name: amBoid.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ****************************************************

#ifndef _amBoid
#define _amBoid


// ****************************************************
//    INCLUDES AND EXTERNS
// ****************************************************

// ****************************************************
//    DEFINES
// ****************************************************


// ****************************************************
//    FUNCTION PROTOTYPE SPECIFICATIONS
// ****************************************************

void initamBoid();

void exitamBoid();


// ****************************************************
//    INLINED MEMBER FUNCTIONS
// ****************************************************


#endif //
```

```
// ****************************************************
//    EXECUTIVE SUMMARY
//
// File Name: amBoid.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ****************************************************

// ****************************************************
//        INCLUDES AND EXTERNS
// ****************************************************
#include "Rti.hh"
#include "amBoid.h"
#include "npsVisual.h"
#include "npsWindow.h"
#include "npsViewport.h"
#include "npsFlyingCamera.h"
#include "bbMouse.h"
#include "bbScreen.h"
#include "bbKeyboard.h"
#include "bbEventResponse.h"
#include "bbCallback.h"
#include "npsGeometry.h"
#include "boid.h"
#include "amHLAAdmin.h"
#include "admin.h"
#include "amObject.h"
#include <math.h>
#include <GL/gl.h>

#ifdef SGI
   #include "bbSGI.h"
#endif

// ****************************************************
//        CODE
// ****************************************************

Boid *myBoid; // global local object

//function prototype
int mouseInWindow();


// ****************************************************
// Function Name: initamBoid()
// Task:  initialize Module -- this is run only once
// Return Value: void
// ****************************************************

void initamBoid() {

    void initKeyboardModule();
```

```
        void initVisualModule();


    myBoid = new Boid();
    Admin::addModuleFunctionPointer(myBoid);

    initKeyboardModule();

    myBoid = NULL;

    cout << "amBoid Module loaded" << endl;
}// end initamBoid


// ****************************************************************
// Function Name: exitamBoid()
// Task:  does housekeeping required to exit the  Module --
//    this is run only once
// Return Value: void
// ****************************************************************

void exitamBoid(){

    bbCallbackHandler *callbackHandler;
    uint               numCallbacks;
    uint               currCallback;
    bbCallback        *callback;
    uint               currDuck;
    npsGeometry       *duck;
    bbKeyboard*        keyboard;
    bbEventResponse*   eventResponse;


    // remove keyboard callbacks
    // RESET
    eventResponse = bbEventResponse::findObject("amBoidER_Reset");
    callback = bbCallback::findObject("amBoidReset");
    eventResponse->removeCallback(callback);
    delete callback;


    // remove keyboard callbacks
    // LoadBoid
    eventResponse = bbEventResponse::findObject("amBoidER_LoadBoid");
    callback = bbCallback::findObject("amBoidLoadBoid");
    eventResponse->removeCallback(callback);
    delete callback;


    // first remove update callback
    callback = npsVisual::appCallback();
    callbackHandler = callback->getPreCallbackHandler();
    callback = bbCallback::findObject("BoidPreApp");
    if(callback){
        callbackHandler->removeCallback(callback);
        delete callback;
    }//end if
```

```
        // remove remote amObjects
        amObject* theObject = Admin::findSimEntity("amBoid");
        while(theObject){
            Admin::removeAmObject(theObject->getEntityID());
            theObject = Admin::findSimEntity("amBoid");
        }

        // remove the module from module list
        Admin::removeModule("amBoid");

}//end exitamBoid


// ****************************************************************
// Function Name: initKeyboardModule()
// Task:   add keyboard callbacks
// Return Value: void
// ****************************************************************

void initKeyboardModule() {

    void escFunc(bbObject *object, bbData *data);
    void resetFunc(bbObject *object, bbData *data);
    void loadBoid(bbObject *object, bbData *data);

    bbKeyboard             *keyboard;
    bbEventResponse        *eventResponse;
    bbCallback             *callback;

    // get the keyboard device
    keyboard = bbKeyboard::getInstance();

    // set up reset key
    eventResponse = new bbEventResponse(bbKeyboard::KEY_SPACE);
    eventResponse->setName("amBoidER_Reset");
    callback = new bbCallback();
    callback->setFunc(resetFunc);
    callback->setName("amBoidReset");
    eventResponse->addCallbackLast(callback);
    keyboard->addEventResponse(eventResponse);

    // set up load boid key
    eventResponse = new bbEventResponse(bbKeyboard::KEY_B |
bbKeyboard::DOWN_TRANS);
    eventResponse->setName("amBoidER_LoadBoid");
    callback = new bbCallback();
    callback->setFunc(loadBoid);
    callback->setName("amBoidLoadBoid");
    eventResponse->addCallbackLast(callback);
    keyboard->addEventResponse(eventResponse);


}//end initKeyboardModule


// ****************************************************************
```

```
// Function Name: initVisualModule()
// Task:  attache callback to PreApp callback handler in Visula Module
// Return Value: void
// ****************************************************************

void initVisualModule(){

    void initCheckerboardFunc(bbObject *object, bbData *data);
    void preAppFunc(bbObject *object, bbData *data);

    npsWindow               *window;
    npsViewport             *viewport;
    npsCamera               *camera;
    bbEventResponse         *eventResponse;
    bbCallbackHandler       *callbackHandler;
    bbCallback              *callback;
    npsVec3f                position;
    npsQuaternion           rotation;
    npsGeometry             *checkerboard;


    // pre app callback
    callback = npsVisual::appCallback();
    callbackHandler = callback->getPreCallbackHandler();
    callback = new bbCallback();
    callback->setName("BoidPreApp");
    callback->setFunc(preAppFunc);
    callbackHandler->addCallbackLast(callback);


}  // end initVisualModule


// ****************************************************************
// Function Name: preAppFunc(bbObject *object, bbData *data)
// Task:  defines preApp callback function --
//    uses keyboard controls to maneuver Boid
//    checks for collisions
// Return Value: void
// ****************************************************************

void preAppFunc(bbObject *object, bbData *data) {

    void deleteMyBoid();

    float           cosh, sinh, cosp, sinp, cosr, sinr;
    bbKeyboard      *kd;
    npsCamera       *camera;
    npsVec3f                position;
    npsQuaternion           rotation;
    npsVec3f                tmpVec;
    float                   hpr[3];
    float                   velocity = 0;

    kd = bbKeyboard::getInstance();
    camera = npsCamera::findObject("AdminCamera");
```

```
    //don't do anything unless there is a local boid to control
    if(myBoid){

        position = myBoid->getPosition();
        rotation = myBoid->getOrientation();
        rotation.getEulers(hpr);

        // Need this to control each window separtely
        // key commands won't work unless the mouse pointer
        // is in the window

        if (mouseInWindow()){

            // update rotation
            if ( kd->getVal(bbKeyboard::KEY_LEFTARROW) ) { // heading left?

                hpr[0] += NPS_DEG2RAD(4.0f);
                if (hpr[0] > NPS_PI)
                    hpr[0] -= 2.0f*NPS_PI;
            }
            if ( kd->getVal(bbKeyboard::KEY_RIGHTARROW) ) { // heading right?

                hpr[0] -= NPS_DEG2RAD(4.0f);
                if (hpr[0] < -NPS_PI)
                    hpr[0] += 2.0f*NPS_PI;
            }
            if ( kd->getVal(bbKeyboard::KEY_UPARROW) ){  // pitching up?

                hpr[1] += NPS_DEG2RAD(4.0f);
                if (hpr[1] > NPS_PI*0.495f)
                    hpr[1] = NPS_PI*0.495f;
            }
            if ( kd->getVal(bbKeyboard::KEY_DOWNARROW) ){  // pitching down?

                hpr[1] -= NPS_DEG2RAD(4.0f);
                if (hpr[1] < -NPS_PI*0.495f)
                    hpr[1] = -NPS_PI*0.495f;
            }

            rotation.setEulers(hpr); // set quaternion

            // update velocity
            if ( kd->getVal(bbKeyboard::KEY_S) && kd->getVal(bbKeyboard::KEY_A)
) // dead stop
                velocity = 0.0f;
            else if ( kd->getVal(bbKeyboard::KEY_S) ) // accelerate
                velocity += 1.2f;
            else if ( kd->getVal(bbKeyboard::KEY_A) ) // decelerate
                velocity -= 1.2f;
            else {                          // slow down

                if (velocity >= 0.1f)
                    velocity -= 0.1f;
                else if (velocity <= -0.1f)
                    velocity += 0.1f;
```

59

```
            else
                velocity = 0.0f;
        }
        if (velocity >= 2.0f)              // clamp velocity -2 <= velocity <=
2
            velocity = 2.0f;
        else if (velocity <= -2.0f)
            velocity = -2.0f;


    }// end if mouseInWindow


    // update position
    static RTI::ObjectID oid = 0;
    oid = 0;

    // entity collision
    oid = myBoid->hasCollided();
    float yPos = 0.0;
    RTI::Boolean terrainFeatureCollision = RTI::RTI_FALSE;

    // check collision with terrain
    if(Admin::ms_Terrain){
        yPos = Admin::ms_Terrain->checkTerrainCollision(myBoid);
        terrainFeatureCollision = Admin::ms_Terrain->checkCollision(myBoid);
    }

    // terrain is returning a y Position
    if(yPos != 0.0){

        tmpVec.set(0.0f, 0.0f, -1.0f);  // forward
        tmpVec.scale(velocity);
        rotation.xform(tmpVec);
        position.add(tmpVec); // set vec3f
        position.set(1,yPos);

        myBoid->setVelocity(tmpVec);
        myBoid->setPosition(position);
        myBoid->setOrientation(rotation);
        camera->setPosition(position);
        camera->setOrientation(rotation);

    }//end if
    // boid collided with terrain feature
    else if(terrainFeatureCollision){

        tmpVec.set(myBoid->getVelocity());  // forward
        // tmpVec.scale(velocity);
        rotation.xform(tmpVec);
        position.sub(tmpVec); // set vec3f
        position.sub(tmpVec);
        position.sub(tmpVec); // set vec3f
        position.sub(tmpVec);
        position.sub(tmpVec); // set vec3f

        myBoid->setVelocity(tmpVec);
        myBoid->setPosition(position);
```

```
            myBoid->setOrientation(rotation);
            camera->setPosition(position);
            camera->setOrientation(rotation);

        }

        // boid no collision
        else if(oid == 0){
            tmpVec.set(0.0f, 0.0f, -1.0f);  // forward
            tmpVec.scale(velocity);
            rotation.xform(tmpVec);
            position.add(tmpVec); // set vec3f

            myBoid->setVelocity(tmpVec);
            myBoid->setPosition(position);
            myBoid->setOrientation(rotation);
            camera->setPosition(position);
            camera->setOrientation(rotation);

        }//end else if
        // boid collided with entity
        else{
            tmpVec.set(0.0f, 0.0f, -1.0f);  // forward
            tmpVec.scale(velocity);
            rotation.xform(tmpVec);
            position.sub(tmpVec); // set vec3f
            position.sub(tmpVec);

            myBoid->setVelocity(tmpVec);
            myBoid->setPosition(position);
            myBoid->setOrientation(rotation);
            camera->setPosition(position);
            camera->setOrientation(rotation);

            cout << "Collided with entity " << oid <<  endl;

            RTI::FederationTime theTime = 0.0;
            myBoid->sendCollisionInteraction(theTime,oid);
        }//end else

        // check if Boid is destroyed in some way
        if(myBoid->getDamage() < 0){
            deleteMyBoid();
        }

    }// end if myBoid

}// end preAppFunc


// *************************************************************
// Function Name: resetFunc(bbObject *object, bbData *data)
// Task:  function defined for the resetBoid keyboard callback
// Return Value: void
// *************************************************************

void resetFunc(bbObject *object, bbData *data){
```

```
    npsCamera*              camera;
    npsVec3f                initPosition;
    npsQuaternion           initRotation;

    if (mouseInWindow() && myBoid){

        initPosition.set(0.0f, 3.0f, -10.0f);
        initRotation.setEulers(NPS_DEG2RAD(180.0f),0.0f,0.0f);

        myBoid->setPosition(initPosition);
        myBoid->setOrientation(initRotation);
    }// end if
} // end resetFunc

// *************************************************************
// Function Name: loadBoid(bbObject *object, bbData *data)
// Task:  function defined for the loadBoid keyboard callback
// Return Value: void
// *************************************************************

void loadBoid(bbObject *object, bbData *data){

    void initVisualModule();

    if (mouseInWindow()){
        cout << "loadBoid" << endl;
        if(!myBoid){
            initVisualModule();
            RTI::ObjectID oid = Admin::registerObject();
            amObject* tmp =
                Admin::addSimEntity("amBoid", oid);
            myBoid = (Boid*)tmp;
            myBoid->setLocalObject(RTI::RTI_TRUE);
        }//end if
    }//end if
}// end loadBoid


// *************************************************************
// Function Name: mouseInWindow()
// Task:  tells if mouse is in the correct window to receive keyboard
//        commands
// Return Value: int representing boolean
// *************************************************************

int mouseInWindow() {
    int flag = 0;

    npsWindow               *window;
    npsViewport             *viewport;
    window = npsWindow::findObject("AdminWindow");
    viewport = npsViewport::findObject("AdminViewport");
    float        x,y;
    x = bbMouse::getX();
    y = bbMouse::getY();
```

```
        bbScreen::normalizeVal(&x,&y);

        if (viewport->getWindow()->isValInside(x,y)){
            flag = 1;
        }
        return flag;
}//end mouseInwindow


void deleteMyBoid() {
    bbCallbackHandler       *callbackHandler;
    bbCallback              *callback;

    Admin::removeAmObject(myBoid->getEntityID());
    myBoid = NULL;


}//end deleteMyBoid
```

```
// ........................................................
//    EXECUTIVE SUMMARY
//
// File Name: Boid.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines boid class
//
// September 1998, Masters Thesis
// ........................................................

#include "Rti.hh"
#include "npsVec3f.h"
#include "npsGeometry.h"
#include "bbKeyboard.h"
#include "amObject.h"
#include "admin.h"

#ifndef ADMIN_API
  #ifdef VISUALCPP
    #define ADMIN_API __declspec(dllimport)
  #else
    #define ADMIN_API
  #endif
#endif


class ADMIN_API Boid : public amObject
{

//memeber variables
public:
    ACE_Recursive_Thread_Mutex theMutex;

    // time management data memebers
    static unsigned int   ms_extentCardinality;
    static RTI::Boolean   ms_timeAdvGrant;
    static RTI::FederationTime  ms_grantTime;
    static char* ms_ModuleName;


private:

    RTI::FederationTime   m_lastTime;

    npsGeometry* myGeom;

    // Change flags for attibute values;

    RTI::Boolean          hasPositionChanged;
    RTI::Boolean          hasRotationChanged;

    RTI::Boolean          localFlag;


    // Update Control flags
```

```
    static RTI::Boolean       ms_sendRotationUpdates;
    static RTI::Boolean       ms_sendPositionUpdates;
    static RTI::Boolean       ms_sendCommInteractions;

// member functions
public:
    Boid();
    Boid( RTI::ObjectID  id);
    void setLocalObject(RTI::Boolean);
    RTI::ObjectID hasCollided();

    virtual ~Boid();

    virtual void display();

    virtual amObject* createObject(RTI::ObjectID oid);

    virtual void deleteObject();

    virtual char* getModuleName();

    virtual RTI::Boolean localObject();

    virtual RTI::Boolean isTerrain();

    // Methods acting on the RTI


    virtual void     sendUpdates(RTI::FederationTime newTime);

    virtual void
sendUpdates(RTI::ObjectID,RTI::AttributeHandleValuePairSet&,
        RTI::FederationTime, RTI::UserSuppliedTag);

    virtual void     receiveUpdates(RTI::ObjectID oid,
            const RTI::AttributeHandleValuePairSet& hvps,
        RTI::FederationTime ft, RTI::UserSuppliedTag tag);

    void     sendCollisionInteraction(RTI::FederationTime
newTime,RTI::ObjectID oid);

    virtual void     sendInteraction(RTI::InteractionClassHandle,
        RTI::ParameterHandleValuePairSet&, RTI::FederationTime,
RTI::UserSuppliedTag);

    virtual void receiveInteraction( RTI::InteractionClassHandle
theInteraction,
            const RTI::ParameterHandleValuePairSet& theParameters );

    virtual void receiveInteraction(RTI::InteractionClassHandle ti,
        RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
RTI::UserSuppliedTag tag);

    virtual RTI::Boolean checkCollision(amObject* that);

    virtual float checkTerrainCollision(amObject* that);
```

```
    // Accessor Methods

    // Static Accessor Methods

    RTI::FederationTime              GetLastTime()
                                        { return m_lastTime; };

private:
    void initBoidGeometry();
    static void initBoidFunc(bbObject *object, bbData *data);
    float distanceFrom(npsVec3f thatPos);
protected:
    RTI::AttributeHandleValuePairSet*   CreateNVPSet();
};
```

```
// ........................................................
//    EXECUTIVE SUMMARY
//
// File Name: Boid.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: method definitions for the Boid class --
//    Boid class represents the class that is implemented by the amArena
//    module.
//
// September 1998, Masters Thesis
// ........................................................


#include "RTI.hh"
#include "boid.h"
#include "npsGeometry.h"
#include "bbKeyboard.h"
#include <GL/gl.h>
#include "admin.h"
#include "amObject.h"
#include "amHLAAdmin.h"


    // Extent data memebers
    unsigned int    Boid::ms_extentCardinality = 0;

    char* Boid::ms_ModuleName = "amBoid";

///////////////////////////////////////////////////////////////////
// Construction/Destruction
///////////////////////////////////////////////////////////////////


// ........................................................
// Function Name: Boid( RTI::ObjectID  id)
// Task:  constructor -- instance intializer
// Return Value: void
// ........................................................

Boid::Boid( RTI::ObjectID  id)
    :m_lastTime(0.0){

    setEntityID(id);
    npsVec3f xx;
    xx.set(0.0f,0.0f,0.0f);
    setPosition(xx);
    npsQuaternion yy;
    setOrientation(yy);

    setDamage(10.0);

    localFlag = RTI::RTI_FALSE;

    initBoidGeometry();
    Boid::ms_extentCardinality++;
```

```
}//end constructor

// ****************************************************************
// Function Name: Boid( )
// Task:  constructor -- used for object storage in ModuleArray
// Return Value: void
// ****************************************************************

Boid::Boid()
    : m_lastTime(0.0){

}


// ****************************************************************
// Function Name: ~Boid()
// Task:  destructor
// Return Value: void
// ****************************************************************

Boid::~Boid(){

    // remove the Object from the

    Boid::ms_extentCardinality--;
    if(myGeom)
        delete myGeom;

}// end destructor


// ****************************************************************
// Function Name: sendUpdates(RTI::ObjectID oid,
//    RTI::AttributeHandleValuePairSet& ta,
//    RTI::FederationTime ft, RTI::UserSuppliedTag tag)
// Task:  not implemented in this module -- pure virtual function must
//    have definition
// Return Value: void
// ****************************************************************

void Boid::sendUpdates(RTI::ObjectID oid,
        RTI::AttributeHandleValuePairSet& ta,
        RTI::FederationTime ft, RTI::UserSuppliedTag tag)
{}


// ****************************************************************
// Function Name: sendUpdates(RTI::FederationTime newTime)
// Task:  send handle value pair to rti for update to federation
// Return Value: void
// ****************************************************************

void Boid::sendUpdates(RTI::FederationTime newTime){

    //-------------------------------------------------------
    // Update state of Boid
```

```
    //-------------------------------------------------------
    RTI::AttributeHandleValuePairSet* pNvpSet = CreateNVPSet();

    Admin::sendEntityUpdate( getEntityID(),*pNvpSet, getModuleName() );


    // set m_lastTime to newTime
    m_lastTime = newTime;

}// end sendUpdates


// ****************************************************************
// Function Name: receiveUpdates( RTI::ObjectID oid,
//    const RTI::AttributeHandleValuePairSet& theAttributes,
//    RTI::FederationTime theTime, RTI::UserSuppliedTag theTag)
// Task:  decodes HVP from RTI for this object
// Return Value: void
// ****************************************************************

void Boid::receiveUpdates( RTI::ObjectID oid,
        const RTI::AttributeHandleValuePairSet& theAttributes,
        RTI::FederationTime theTime, RTI::UserSuppliedTag theTag){

    RTI::AttributeHandle attrHandle;
    unsigned long        valueLength;

    // We need to iterate through the AttributeHandleValuePairSet
    // to extract each AttributeHandleValuePair.  Based on the type
    // specified ( the value returned by getHandle() ) we need to
    // extract the data frlom the buffer that is returned by
    // getValue().
    for ( unsigned int i = 0; i < theAttributes.size(); i++ )
    {
        attrHandle = theAttributes.getHandle( i );
        if ( attrHandle == Admin::getPositionAttrHandle() )
        {
            npsVec3f tmp;
            theAttributes.getValue( i, (char*)&tmp, valueLength );
            setPosition(tmp);
        }//end if

        else if ( attrHandle == Admin::getOrientationAttrHandle() )
        {
            // Same as above goes here...
            npsQuaternion tmp;
            theAttributes.getValue( i, (char*)&tmp, valueLength );
            setOrientation(tmp);
        }// end else if

        else if ( attrHandle == Admin::getVelocityAttrHandle() )
        {
            // Same as above goes here...
            npsVec3f tmp;
            theAttributes.getValue( i, (char*)&tmp, valueLength );
            setVelocity(tmp);
        }// end else if
```

```
    }// end for

}//end recevieUpdates


// ****************************************************************
// Function Name: receiveInteraction(RTI::InteractionClassHandle ich,
//    RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
//    RTI::UserSuppliedTag tag)
// Task:  not implemented in this module -- pure virtual function must
//    have definition
// Return Value: void
// ****************************************************************

void Boid::receiveInteraction(RTI::InteractionClassHandle ich,
        RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
        RTI::UserSuppliedTag tag)
{}


// ****************************************************************
// Function Name: receiveInteraction(
//    RTI::InteractionClassHandle theInteraction,
//    const RTI::ParameterHandleValuePairSet& theParameters )
// Task:  recieves PHVP and decodes for use with module
// Return Value: void
// ****************************************************************

void Boid::receiveInteraction( RTI::InteractionClassHandle theInteraction,
            const RTI::ParameterHandleValuePairSet& theParameters ){

    if(theInteraction == Admin::getCollisionInteractionHandle()){
        setDamage(getDamage() - 1.0);

        cout << "**Collision Detected -- Object " << getEntityID()
            << " Damage = " << getDamage() << endl;
    }

}//end receiveInteraction


// ****************************************************************
// Function Name: sendCollisionInteraction(RTI::FederationTime theTime,
//                                          RTI::ObjectID oid)
// Task:  sends rti the PHVP using the send interaction fuction
// Return Value: void
// ****************************************************************

void Boid::sendCollisionInteraction(RTI::FederationTime theTime,
                                    RTI::ObjectID oid){

    //build paramter handle value pair
    RTI::ParameterHandleValuePairSet* params =
RTI::ParameterSetFactory::create(1);
```

```
    params->add(Admin::getEntityIdHandle(), (char*)&oid,
sizeof(RTI::ObjectID));

    sendInteraction(Admin::getCollisionInteractionHandle(),*params,
            theTime,NULL);

}//end sendCollisionInteraction


// ****************************************************************
// Function Name: sendInteraction(RTI::FederationTime ft)
// Task:  send the PHVP for the given interaction
// Return Value: void
// ****************************************************************

void Boid::sendInteraction(RTI::InteractionClassHandle ihc,
        RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
        RTI::UserSuppliedTag tag){

    try{
        Admin::ms_rtiAmb->sendInteraction(ihc,phvps,ft,tag);
    }
    catch(RTI::Exception& e){
        cerr << &e << endl;
    }

}//end sendInteraction


// ****************************************************************
// Function Name: CreateNVPSet()
// Task:  creates HVP set pertinent to this object
// Return Value: AttributeHandleValuePairSet*
// ****************************************************************

RTI::AttributeHandleValuePairSet* Boid::CreateNVPSet(){

    RTI::AttributeHandleValuePairSet* pBoidAttributes = NULL;

    // Make sure the RTI Ambassador is set.
    if ( Admin::ms_rtiAmb )
    {
        //-------------------------------------------------------
        // Set up the data structure required to push this
        // objects's state to the RTI.
        //-------------------------------------------------------
        RTI::ObjectIDcount numAttributes(3);
        pBoidAttributes = RTI::AttributeSetFactory::create( numAttributes );

        pBoidAttributes->add( Admin::getOrientationAttrHandle(),
                                (char*) &orientation,
                                sizeof(npsQuaternion) );

        pBoidAttributes->add( Admin::getPositionAttrHandle(),
                                (char*)&position,
                                sizeof(npsVec3f) );
```

```cpp
        pBoidAttributes->add( Admin::getVelocityAttrHandle(),
                              (char*)&velocity,
                              sizeof(npsVec3f) );

    }

    return pBoidAttributes;

}// CreateNVPSet()


// **************************************************************
// Function Name: display()
// Task:  updates geometry position and orientation
// Return Value: void
// **************************************************************

void Boid::display(){
    myGeom->setPosition(getPosition());
    myGeom->setOrientation(getOrientation());

}// end display


// **************************************************************
// Function Name: createObject(RTI::ObjectID oid)
// Task:  creates an object and returns a pointer to it
// Return Value: amObject*
// **************************************************************

amObject* Boid::createObject(RTI::ObjectID oid){
    Boid* tmpBoid = new Boid(oid);
    return (amObject*)tmpBoid;
}//end createObject


// **************************************************************
// Function Name: deleteObject()
// Task:  remote delete function
// Return Value: void
// **************************************************************

void Boid::deleteObject(){
    delete this;
}


// **************************************************************
// Function Name: getModuleName()
// Task:  accessor method for module name
// Return Value: char*
// **************************************************************

char* Boid::getModuleName(){
    return Boid::ms_ModuleName;
}// getModuleName
```

```cpp
// **************************************************************
// Function Name: initBoidFunc(bbObject *object, bbData *data)
// Task:  defines function for geometry callback
// Return Value: void
// **************************************************************

void Boid::initBoidFunc(bbObject *object, bbData *data){

    GLfloat coords[4][3] = { ( 0.0f,  0.0f, -0.9f),  // front
                             (-0.6f,  0.0f,  0.9f),  // back left
                             ( 0.6f,  0.0f,  0.9f),  // back right
                             ( 0.0f,  0.6f,  0.9f)   // top
                           };

    glShadeModel(GL_FLAT);
    glBegin(GL_TRIANGLES);
    {
        glColor3f(0.0f, 0.5f, 1.0f);    // bottom
        glVertex3fv(coords[0]);
        glVertex3fv(coords[2]);
        glVertex3fv(coords[1]);

        glColor3f(1.0f, 0.5f, 0.0f);    // left
        glVertex3fv(coords[0]);
        glVertex3fv(coords[1]);
        glVertex3fv(coords[3]);

        glColor3f(1.0f, 0.5f, 1.0f);    // right
        glVertex3fv(coords[0]);
        glVertex3fv(coords[3]);
        glVertex3fv(coords[2]);

        glColor3f(0.0f, 0.5f, 0.0f);    // back
        glVertex3fv(coords[1]);
        glVertex3fv(coords[2]);
        glVertex3fv(coords[3]);
    }
    glEnd();
    glShadeModel(GL_SMOOTH);
}//end initBoidFunc


// **************************************************************
// Function Name: initBoidGeometry()
// Task:  attaches the function for geometry callback to the Visual Module
// Return Value: void
// **************************************************************

void Boid::initBoidGeometry(){

    myGeom = new npsGeometry(Boid::initBoidFunc);
    myGeom->setBoundingSphere(npsVec4f(0.0f, 0.0f, 0.0f, 1.0f));

}// end initBoidGeometry
```

```cpp
// **************************************************************
// Function Name: setLocalObject(RTI::Boolean flag)
// Task:  set the object as local or not
// Return Value: void
// **************************************************************

void Boid::setLocalObject(RTI::Boolean flag){
    localFlag = flag;
}


// **************************************************************
// Function Name: localObject()
// Task:  accessor method for local object flag
// Return Value: Boolean
// **************************************************************

RTI::Boolean Boid::localObject(){
    return localFlag;
}


// **************************************************************
// Function Name: isTerrain()
// Task:  informs amHLAAdmin whether the object is a terrain object or not
// Return Value: Boolean
// **************************************************************

RTI::Boolean Boid::isTerrain(){
    return RTI::RTI_FALSE;
}


// **************************************************************
// Function Name: checkCollision(amObject* that)
// Task:  returnds true if there is a collision with this object or not
// Return Value: Boolean
// **************************************************************

RTI::Boolean Boid::checkCollision(amObject* that){
    RTI::Boolean flag = RTI::RTI_FALSE;
    if (that != this){
        if(distanceFrom(that->getPosition()) < 2){
            flag = RTI::RTI_TRUE;
        }//end if
    }//end if
    return flag;
}//end checkCollision


// **************************************************************
// Function Name: hasCollided()
// Task:  checks if current object is collided with any other obaject
// Return Value: ObjectID
// **************************************************************

RTI::ObjectID Boid::hasCollided(){
```

```cpp
    // loop amObject array
    // compute distance
    RTI::ObjectID oid = 0;
    amObject* tmp = Admin::checkEntityCollision(this);

    if (tmp && (tmp != this) ){
        if(((tmp->getVelocity()).length() < (this->getVelocity()).length())){
            oid = tmp->getEntityID();
        }//end if
    }//end if

    return oid;
}//end hasCollided


// **************************************************************
// Function Name: distanceFrom(npsVec3f thatPos)
// Task:  finds distance between two objects
// Return Value: float
// **************************************************************

float Boid::distanceFrom(npsVec3f thatPos){

    npsVec3f thisPos = this->getPosition();
    npsVec3f tmpPos;
    tmpPos.sub(thatPos,thisPos);

    return tmpPos.length();

}//end distanceFrom


// **************************************************************
// Function Name: checkTerrainCollision(amObject* that)
// Task:  returns y value for terrain to provide rudimentary terrain
//        collision detection -- 0.0 because not a terrain module
// Return Value: void
// **************************************************************

float Boid::checkTerrainCollision(amObject* that){
    return 0.0;
}//end checkTerrainCollision
```

amArena Files

This is the module.txt file for amArena module

```
Bamboo1.0b
3
bbKeyboardModule
npsVisualModule
amHLAAdmin
```

```
// ***************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: module.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: This file defines the global functions required by every
//              dynamically-linked library.  How these functions are
//              implemented is arbitrary, but it is useful have RCS
//              automate some of the work.
//
// September 1998, Masters Thesis
// ***************************************************************


#ifndef __module
#define __module


// ***************************************************************
//    INCLUDES AND EXTERNS
// ***************************************************************


// ***************************************************************
//    DEFINES
// ***************************************************************


// ***************************************************************
//    FUNCTION PROTOTYPE SPECIFICATIONS
// ***************************************************************


#ifdef __cplusplus
extern "C" {
#endif

  ADMIN_API const char *getModuleName();
  ADMIN_API float getModuleVersion();
  ADMIN_API const char *getModuleDate();
  ADMIN_API const char *getModuleText();
  ADMIN_API bool initModule();
  ADMIN_API bool exitModule();

#ifdef __cplusplus
};
#endif

#endif // __module
```

```
// ***************************************************************
//    EXECUTIVE SUMMARY
//
// File Name: module.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: This file defines the global functions required by every
//              dynamically-linked library.  How these functions are
//              implemented is arbitrary, but it is useful have RCS
//              automate some of the work.
//
// September 1998, Masters Thesis
// ***************************************************************


// ***************************************************************
//    INCLUDES AND EXTERNS
// ***************************************************************

#include <stdio.h>
#include <stdlib.h> // atof

#include "module.h"
#include "amArena.h"


// ***************************************************************
//    DEFINES & FILE SCOPE VARIABLES
// ***************************************************************


// ***************************************************************
//    CODE
// ***************************************************************


const char *getModuleName()
  {
  return "amArena";
  }


float getModuleVersion()
  {
  return 1.0f;
  }


const char *getModuleDate()
  {
  return "1998/08/01 06:00:48";
  }


const char *getModuleText()
```

```
        {
        return "amArena Terrain module -- T to load";
        }


bool initModule()
        {
        initamArena();
        return 1;
        }


bool exitModule()
        {
        exitamArena();
        return 1;
        }
```

```
// ...........................................................
//    EXECUTIVE SUMMARY
//
// File Name: amArena.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ...........................................................


// ...........................................................
//         INCLUDES AND EXTERNS
// ...........................................................

#include "Rti.hh"
#include "npsVisual.h"
#include "npsWindow.h"
#include "npsViewport.h"
#include "bbMouse.h"
#include "bbKeyboard.h"
#include "bbEventResponse.h"
#include "bbCallback.h"
#include "npsGeometry.h"
#include "Arena.h"
#include "amHLAAdmin.h"
#include "admin.h"
#include "amObject.h"


#include <math.h>
#include <GL/gl.h>

#ifdef SGI
   #include "bbSGI.h"
#endif

// ...........................................................
//         CODE
// ...........................................................

Arena *myArena;

int mouseInWindow();


// ...........................................................
// Function Name: initamArena()
// Task:  initialize Module -- this is run only once
// Return Value: void
// ...........................................................

void initamArena() {
```

```
    // prototypes
    void initKeyboardModule();
    void initVisualModule();

    // instance a Arena Object and pass it to the module array
    myArena = new Arena();
    Admin::addModuleFunctionPointer(myArena);

    //this is a Terrain module so if one already exists we must delete it
    if (Admin::ms_Terrain){
        Admin::unloadModule(Admin::ms_Terrain->getModuleName());
    }//end if

    initKeyboardModule();

    // null my Arena for future use
    myArena = NULL;

    // add object to the federation
    if(!myArena){
        RTI::ObjectID oid = Admin::registerObject();
        amObject* tmp = Admin::findModule("amArena");
        npsGeometry* arenaPtr = npsGeometry::findObject("amArena_Geom");
        if (!arenaPtr){
            initVisualModule();
        }
        myArena = (Arena*)tmp->createObject(oid);
        myArena->setLocalObject(RTI::RTI_TRUE);
        Admin::ms_Terrain = myArena;
    }//end if

    // this allows Arena to update to rest of federation
    if(myArena){
        myArena->setUpdateFlag(RTI::RTI_TRUE);
    }

    cout << "initamArena" << endl;

}// end initamArena


// ...........................................................
// Function Name: exitamArena()
// Task:  does housekeeping required to exit the  Module --
//    this is run only once
// Return Value: void
// ...........................................................

void exitamArena(){
    bbCallbackHandler *callbackHandler;
    bbCallback        *callback;
    bbKeyboard*        keyboard;
    bbEventResponse    eventResponse;

    keyboard = bbKeyboard::getInstance();

    // remove keyboard callbacks
```

```
    // LoadArena callback
    eventResponse = bbEventResponse::findObject("amArenaER_Terrain");
    callback = bbCallback::findObject("amArena_Terrain");
    eventResponse->removeCallback(callback);
    delete callback;

    // remove remote amObjects
    if(Admin::ms_Terrain){
        Admin::removeAmObject(Admin::ms_Terrain->getEntityID());
    }

    //remove local object
    if(!myArena){
        npsGeometry* tmp = npsGeometry::findObject("amArena_Geom");
        if(tmp)
            delete tmp;
    }//end else

    // remove the module from module list
    Admin::removeModule("amArena");

    Admin::ms_Terrain = NULL;

    cout << "exitamArena" << endl;

} //end exitamArena


// ************************************************************
// Function Name: initKeyboardModule()
// Task:  add keyboard callbacks
// Return Value: void
// ************************************************************

void initKeyboardModule()
    {

    void escFunc(bbObject *object, bbData *data);
    void resetFunc(bbObject *object, bbData *data);
    void loadArena(bbObject *object, bbData *data);

    bbKeyboard              *keyboard;
    bbEventResponse         *eventResponse;
    bbCallback              *callback;

    // get the keyboard device
    keyboard = bbKeyboard::getInstance();


    // set up load boid key
    eventResponse = new bbEventResponse(bbKeyboard::KEY_T |
bbKeyboard::DOWN_TRANS);
    eventResponse->setName("amArenaER_Terrain");
    callback = new bbCallback();
    callback->setName("amArena_Terrain");
    callback->setFunc(loadArena);
    eventResponse->addCallbackLast(callback);
```

```
    }


// ************************************************************
// Function Name: initArenaFunc(bbObject *object, bbData *data)
// Task:  function defining the Arena geometry
// Return Value: void
// ************************************************************

void initArenaFunc(bbObject *object, bbData *data){
    npsGeometry       *geometry;
    uint                       displayListNum;
    const float     CELL_LENGTH = 5.0;
    #ifdef VISUALCPP
        const uint      NUM_CELLS_LONG = 10;
        const uint      NUM_CELLS_WIDE = 10;
    #elif SGI
        const uint      NUM_CELLS_LONG = 50;
        const uint      NUM_CELLS_WIDE = 50;
    #endif
    const uint      TOTAL_NUM_CELLS = NUM_CELLS_LONG * NUM_CELLS_WIDE;
    const uint      NUM_VERTS_LONG = NUM_CELLS_LONG + 1;
    const uint      NUM_VERTS_WIDE = NUM_CELLS_WIDE + 1;
    const uint      TOTAL_NUM_VERTS = NUM_VERTS_LONG * NUM_VERTS_WIDE;
    uint            i, j, currVert, currIndex, currCell;
    bool            colorToggle;

    GLfloat           coords[TOTAL_NUM_VERTS][3];
    GLfloat           normals[TOTAL_NUM_VERTS][3];
    GLfloat           textures[TOTAL_NUM_VERTS][2];


    // init Vals
    colorToggle = 0;
    for (i=0; i<NUM_VERTS_LONG; i++){
        for (j=0; j<NUM_VERTS_WIDE; j++){
            currVert = i*NUM_VERTS_WIDE + j;

            coords[currVert][0] = (CELL_LENGTH * i) -
(NUM_CELLS_LONG*CELL_LENGTH*0.5f);
            coords[currVert][1] = 0.0f;
            coords[currVert][2] = (-CELL_LENGTH * j) +
(NUM_CELLS_WIDE*CELL_LENGTH*0.5f);
            normals[currVert][0] = 0.0f;
            normals[currVert][1] = 1.0f;
            normals[currVert][2] = 0.0f;
            textures[currVert][0] = ((float)i)/(NUM_VERTS_LONG-1);
            textures[currVert][1] = ((float)j)/(NUM_VERTS_WIDE-1);
        }//end for j
    }//end for i

    glShadeModel(GL_FLAT);


    colorToggle = 0;

    for (i=0; i<NUM_CELLS_LONG; i++){
```

```
    keyboard->addEventResponse(eventResponse);


    }

// ************************************************************
// Function Name: initVisualModule()
// Task:  set the geometry for the Arena
// Return Value: void
// ************************************************************

void initVisualModule(){

    void initArenaFunc(bbObject *object, bbData *data);

    npsGeometry              *arena;

    // init geometry
    arena = new npsGeometry(initArenaFunc);
    arena->setName("amArena_Geom");

} // end initVisualModule


// ************************************************************
// Function Name: loadArena(bbObject *object, bbData *data)
// Task:  function defined for the loadArena keyboard callback
// Return Value: void
// ************************************************************

void loadArena(bbObject *object, bbData *data){
    void initVisualModule();

    if (mouseInWindow()){
        cout << "loadArena" << endl;

        if(!myArena){
            //add Arena to federation if does not exist
            RTI::ObjectID oid = Admin::registerObject();
            amObject* tmp = Admin::findModule("amArena");
            npsGeometry* arenaPtr = npsGeometry::findObject("amArena_Geom");
            if (!arenaPtr){
                initVisualModule();
            }
            myArena = (Arena*)tmp->createObject(oid);
            myArena->setLocalObject(RTI::RTI_TRUE);
            Admin::ms_Terrain = myArena;
        }//end if

        //set flag so Arena is updated and the rest of the federation
        // will discover the object
        if(myArena){
            myArena->setUpdateFlag(RTI::RTI_TRUE);
        }
    }//end if
```

```
    glBegin(GL_TRIANGLE_STRIP);
    { //start gl

        for (j=0; j<NUM_VERTS_WIDE; j++){
            if (colorToggle){
                colorToggle = 0;
                glColor3f(0.0f, 0.0f, 1.0f);
            }
            else{
                colorToggle = 1;
                glColor3f(1.0f, 1.0f, 1.0f);
            }
            currVert = i*NUM_VERTS_WIDE + j;
            glVertex3fv(coords[currVert]);

            currVert = (i+1)*NUM_VERTS_WIDE + j;
            glVertex3fv(coords[currVert]);
        }//end for
        if ( NUM_CELLS_WIDE & 0x1 ){
            if (colorToggle)
                colorToggle = 0;
            else
                colorToggle = 1;
        }
    }//end gl
    glEnd();
}//end for

// walls and ceiling
//North wall
glBegin(GL_POLYGON);
    glColor3f(0.5f,0.5f,0.5f);
    glVertex3f(-25.0f,0.0f,-25.0f);
    glVertex3f(25.0f,0.0f,-25.0f);
    glVertex3f(25.0f,50.0f,-25.0f);
    glVertex3f(-25.0f,50.0f,-25.0f);
glEnd();
//South wall
glBegin(GL_POLYGON);
    glColor3f(0.6f,0.6f,0.6f);
    glVertex3f(-25.0f,0.0f,25.0f);
    glVertex3f(25.0f,0.0f,25.0f);
    glVertex3f(25.0f,50.0f,25.0f);
    glVertex3f(-25.0f,50.0f,25.0f);
glEnd();
//West wall
glBegin(GL_POLYGON);
    glColor3f(0.8f,0.5f,0.4f);
    glVertex3f(-25.0f,0.0f,25.0f);
    glVertex3f(-25.0f,0.0f,-25.0f);
    glVertex3f(-25.0f,50.0f,-25.0f);
    glVertex3f(-25.0f,50.0f,25.0f);
glEnd();
//East wall
glBegin(GL_POLYGON);
    glColor3f(0.4f,0.5f,0.9f);
    glVertex3f(25.0f,0.0f,-25.0f);
```

```
        glVertex3f(25.0f,0.0f,25.0f);
        glVertex3f(25.0f,50.0f,25.0f);
        glVertex3f(25.0f,50.0f,-25.0f);
    glEnd();
    //ceiling wall
    glBegin(GL_POLYGON);
        glColor3f(0.8f,0.8f,0.8f);
        glVertex3f(-25.0f,50.0f,-25.0f);
        glVertex3f(25.0f,50.0f,-25.0f);
        glVertex3f(25.0f,50.0f,25.0f);
        glVertex3f(-25.0f,50.0f,25.0f);
    glEnd();

    glShadeModel(GL_SMOOTH);
}//initArenaFunc


// ************************************************************
// Function Name: mouseInWindow()
// Task:  tells if mouse is in the correct window to receive keyboard
//    commands
// Return Value: int representing boolean
// ************************************************************

int mouseInWindow() {
  int flag = 0;

  npsWindow              *window;
  npsViewport            *viewport;
  window = npsWindow::findObject("AdminWindow");
  viewport = npsViewport::findObject("AdminViewport");
  float                  x,y;
  x = bbMouse::getX();
  y = bbMouse::getY();
  bbScreen::normalizeVal(&x,&y);

  if (viewport->getWindow()->isValInside(x,y)){
     flag = 1;
  }
  return flag;
}
```

```
        virtual void display();

        virtual amObject* createObject(RTI::ObjectID oid);

        virtual char* getModuleName();

        virtual RTI::Boolean localObject();

        virtual RTI::Boolean isTerrain();

        // Methods acting on the RTI

        virtual void      sendUpdates(RTI::FederationTime newTime);

        virtual void
sendUpdates(RTI::ObjectID,RTI::AttributeHandleValuePairSet&,
        RTI::FederationTime, RTI::UserSuppliedTag);

        virtual void      receiveUpdates(RTI::ObjectID oid,
                const RTI::AttributeHandleValuePairSet& hvps,
            RTI::FederationTime ft, RTI::UserSuppliedTag tag);

        virtual void      sendInteraction(RTI::InteractionClassHandle,
            RTI::ParameterHandleValuePairSet&, RTI::FederationTime,
RTI::UserSuppliedTag);

        virtual void      sendInteraction(RTI::FederationTime);

        virtual void receiveInteraction( RTI::InteractionClassHandle
theInteraction,
                const RTI::ParameterHandleValuePairSet& theParameters );

        virtual void      receiveInteraction(RTI::InteractionClassHandle,
            RTI::ParameterHandleValuePairSet&, RTI::FederationTime,
RTI::UserSuppliedTag);

        virtual RTI::Boolean checkCollision(amObject* that);

        virtual float checkTerrainCollision(amObject* that);

        virtual void deleteObject();


protected:
    RTI::AttributeHandleValuePairSet*   CreateNVPSet();
};
```

```
// ************************************************************
//   EXECUTIVE SUMMARY
//
// File Name: Arena.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: method definitions for the Arena class --
//    Arena class represents the class that is implemented by the amArena
//    module.
//
// September 1998, Masters Thesis
// ************************************************************

#include "Rti.hh"
#include "npsVec3f.h"
#include "npsGeometry.h"
#include "bbKeyboard.h"
#include "amObject.h"
#include "admin.h"

#ifndef ADMIN_API
  #ifdef VISUALCPP
    #define ADMIN_API __declspec(dllimport)
  #else
    #define ADMIN_API
  #endif
#endif


class ADMIN_API Arena : public amObject
{

//memeber variables
public:
    // Extent data memebers
    static char* ms_ModuleName;

private:



    // Change flags for attibute values;

    RTI::Boolean           localFlag;
    RTI::Boolean           updateFlag; // flag so only single update

// member functions
public:

    Arena();
    Arena( RTI::ObjectID  id);
    void setLocalObject(RTI::Boolean);
    void setUpdateFlag(RTI::Boolean);

    virtual ~Arena();
```

```
// ************************************************************
//   EXECUTIVE SUMMARY
//
// File Name: Arena.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: method definitions for the Arena class --
//    Arena class represents the class that is implemented by the amArena
//    module.
//
// September 1998, Masters Thesis
// ************************************************************


#include "RTI.hh"
#include "Arena.h"
#include "npsGeometry.h"
#include "bbKeyboard.h"
#include <GL/gl.h>
#include "admin.h"
#include "amObject.h"
#include "amHLAAdmin.h"


    // Extent data memebers

    char* Arena::ms_ModuleName = "amArena";


///////////////////////////////////////////////////////////////
// Construction/Destruction
///////////////////////////////////////////////////////////////


// ************************************************************
// Function Name: Arena( RTI::ObjectID  id)
// Task:  constructor -- instance intializer
// Return Value: void
// ************************************************************

Arena::Arena( RTI::ObjectID  id){

    cout << "Arena(oid)" << id << endl;

    setEntityID(id);
    npsVec3f xx;
    xx.set(0.0f,0.0f,0.0f);
    setPosition(xx);
    npsQuaternion yy;
    setOrientation(yy);

    localFlag = RTI::RTI_FALSE;
    updateFlag = RTI::RTI_TRUE;

    Admin::ms_Terrain = this;
```

```
}//end constructor

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: Arena( )
// Task:  constructor -- used for object storage in ModuleArray
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

Arena::Arena(){

}//end default constructor

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: ~Arena()
// Task:  destructor
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

Arena::~Arena()
{
    cout << "The arena geometry was just deleted\n"
         << "If you deleted the terrain ignore message\n"
         << "else a federate just quit the federation\n"
         << "PUSH  T to display terrain"
         << endl;
    npsGeometry* tmp = npsGeometry::findObject("amArena_Geom");
    delete tmp;

}//end destructor

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: sendUpdates(RTI::ObjectID oid,
// RTI::AttributeHandleValuePairSet& ta,
//     RTI::FederationTime ft, RTI::UserSuppliedTag tag)
// Task:  not implemented in this module -- pure virtual function must
//     have definition
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::sendUpdates(RTI::ObjectID oid, RTI::AttributeHandleValuePairSet&
ta,
        RTI::FederationTime ft, RTI::UserSuppliedTag tag)
{}

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: sendUpdates(RTI::FederationTime newTime)
// Task:  send handle value pair to rti for update to federation
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::sendUpdates(RTI::FederationTime newTime){
```

```
        if(updateFlag){

            RTI::AttributeHandleValuePairSet* pNvpSet = CreateNVPSet();

            Admin::sendEntityUpdate( getEntityID(),*pNvpSet, getModuleName() );

            updateFlag = RTI::RTI_FALSE;
        }//end if

}// end sendUpdates

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: receiveUpdates( RTI::ObjectID oid,const
RTI::AttributeHandleValuePairSet& theAttributes,
//     RTI::FederationTime theTime, RTI::UserSuppliedTag theTag)
// Task:  decodes HVP from RTI for this object
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::receiveUpdates( RTI::ObjectID oid,const
RTI::AttributeHandleValuePairSet& theAttributes,
        RTI::FederationTime theTime, RTI::UserSuppliedTag theTag){
    RTI::AttributeHandle attrHandle;
    unsigned long        valueLength;

    // We need to iterate through the AttributeHandleValuePairSet
    // to extract each AttributeHandleValuePair.  Based on the type
    // specified ( the value returned by getHandle() ) we need to
    // extract the data frlom the buffer that is returned by
    // getValue().
    for ( unsigned int i = 0; i < theAttributes.size(); i++ )
    {
        attrHandle = theAttributes.getHandle( i );
        if ( attrHandle == Admin::getPositionAttrHandle() )
        {
            npsVec3f tmp;
            theAttributes.getValue( i, (char*)&tmp, valueLength );
            setPosition(tmp);
        }//end if

    }// end for

}//end recevieUpdates

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: receiveInteraction(RTI::InteractionClassHandle ich,
//     RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
//     RTI::UserSuppliedTag tag)
// Task:  not implemented in this module -- pure virtual function must
//     have definition
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
```

```
void Arena::receiveInteraction(RTI::InteractionClassHandle ich,
        RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
        RTI::UserSuppliedTag tag)
{}

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: receiveInteraction( RTI::InteractionClassHandle
theInteraction,
//             const RTI::ParameterHandleValuePairSet& theParameters )
// Task:  not implemented in this module -- pure virtual function must
//     have definition
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::receiveInteraction( RTI::InteractionClassHandle theInteraction,
            const RTI::ParameterHandleValuePairSet& theParameters )
{}//end receiveInteraction

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: sendInteraction(RTI::FederationTime ft)
// Task:  not implemented in this module -- pure virtual function must
//     have definition
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::sendInteraction(RTI::FederationTime ft)
{}

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: sendInteraction(RTI::InteractionClassHandle ihc,
//     RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
//     RTI::UserSuppliedTag tag)
// Task:  not implemented in this module -- pure virtual function must
//     have definition
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::sendInteraction(RTI::InteractionClassHandle ihc,
        RTI::ParameterHandleValuePairSet& phvps, RTI::FederationTime ft,
        RTI::UserSuppliedTag tag)
{}

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: CreateNVPSet()
// Task:  creates HVP set pertinent to this object
// Return Value: AttributeHandleValuePairSet*
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

RTI::AttributeHandleValuePairSet* Arena::CreateNVPSet(){

    RTI::AttributeHandleValuePairSet* pArenaAttributes = NULL;
```

```
    // Make sure the RTI Ambassador is set.
    if ( Admin::ms_rtiAmb )
    {
        //---------------------------------------------------------
        // Set up the data structure required to push this
        // objects's state to the RTI.
        //---------------------------------------------------------
        RTI::ObjectIDcount numAttributes(1);
        pArenaAttributes = RTI::AttributeSetFactory::create( numAttributes );

        pArenaAttributes->add( Admin::getPositionAttrHandle(),
                               (char*)&position,
                               sizeof(npsVec3f) );

    }

    return pArenaAttributes;
}//end CreateNVPSet()

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: display()
// Task:  not implemented in this module -- pure virtual function must
//     have definition
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

void Arena::display(){

}// end display

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: createObject(RTI::ObjectID oid)
// Task:  creates an object and returns a pointer to it
// Return Value: amObject*
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

amObject* Arena::createObject(RTI::ObjectID oid){

    Arena* tmpArena = new Arena(oid);
    return (amObject*)tmpArena;

}//end createObject

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: getModuleName()
// Task:  accessor method for module name
// Return Value: char*
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••••••

char* Arena::getModuleName(){
    return Arena::ms_ModuleName;
}// getModuleName
```

```
// ****************************************************
// Function Name: setLocalObject(RTI::Boolean flag)
// Task:  set the object as local or not
// Return Value: void
// ****************************************************

void Arena::setLocalObject(RTI::Boolean flag){
    localFlag = flag;
}//end setLocalObject


// ****************************************************
// Function Name: localObject()
// Task:  accessor method for local object flag
// Return Value: Boolean
// ****************************************************

RTI::Boolean Arena::localObject(){
    return localFlag;
}//end localObject


// ****************************************************
// Function Name: isTerrain()
// Task:  informs amHLAAdmin whether the object is a terrain object or not
// Return Value: Boolean
// ****************************************************

RTI::Boolean Arena::isTerrain(){
    return RTI::RTI_TRUE;
}//end isTerrain


// ****************************************************
// Function Name: setUpdateFlag(RTI::Boolean flag)
// Task:  sets update flag -- used in send updates to decide whether to
//        transmit a HVP or not
// Return Value: void
// ****************************************************

void Arena::setUpdateFlag(RTI::Boolean flag){
    updateFlag = flag;
}//end setUpdateFlag


// ****************************************************
// Function Name: checkCollision(amObject* that)
// Task:  returnds true if there is a collision with this object or not
// Return Value: Boolean
// ****************************************************

RTI::Boolean Arena::checkCollision(amObject* that){

    RTI::Boolean flag = RTI::RTI_FALSE;

    // floor ckeck
    if((that->getPosition()).get(1) < 0.5f){
```

```
        flag = RTI::RTI_TRUE;
    }
    // ceiling check
    else if((that->getPosition()).get(1) > 49.0f){
        flag = RTI::RTI_TRUE;
    }
    // North wall check
    else if((that->getPosition()).get(2) < -24.0f){
        flag = RTI::RTI_TRUE;
    }
    // south Wall check
    else if((that->getPosition()).get(2) > 24.0f){
        flag = RTI::RTI_TRUE;
    }
    // West wall check
    else if((that->getPosition()).get(0) < -24.0f){
        flag = RTI::RTI_TRUE;
    }
    // East wall check
    else if((that->getPosition()).get(0) > 24.0f){
        flag = RTI::RTI_TRUE;
    }

    return flag;
}//end checkCollision


// ****************************************************
// Function Name: checkTerrainCollision(amObject* that)
// Task:  returns y value for terrain to provide rudimentary terrain
//        collision detection
// Return Value: void
// ****************************************************

float Arena::checkTerrainCollision(amObject* that){
    float yPosition = 0.0;

    if((that->getPosition()).get(1) < 0.5){
        yPosition = 0.5;
    }

    return yPosition;

}//end checkTerrainCollision


// ****************************************************
// Function Name: deleteObject()
// Task:  remote delete function
// Return Value: void
// ****************************************************

void Arena::deleteObject(){

    delete this;

}
```

amPageModule Files

This is the module.txt file for amPageModule module.

Bamboo1.0b

3
bbKeyboardModule
bbMouseModule
npsVisualModule

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    EXECUTIVE SUMMARY
//
// File Name: module.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

#ifndef __module
#define __module

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    FUNCTION PROTOTYPE SPECIFICATIONS
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

#ifdef __cplusplus
extern "C" {
#endif

  ADMIN_API const char *getModuleName();
  ADMIN_API float getModuleVersion();
  ADMIN_API const char *getModuleDate();
  ADMIN_API const char *getModuleText();
  ADMIN_API bool initModule();
  ADMIN_API bool exitModule();

#ifdef __cplusplus
};
#endif

#endif // __module
```

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    EXECUTIVE SUMMARY
//
// File Name: module.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    INCLUDES AND EXTERNS
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

#include "module.h"
#include "amPageModule.h"

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    CODE
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

const char *getModuleName()
  {
  return "amPageModule";
  }

float getModuleVersion()
  {
  return 1.0;
  }

const char *getModuleDate()
  {
  return "1998/09/01 06:05:48";
  }

const char *getModuleText()
  {
  return "This module enables the user to dynamically page modules in\n
         and out of the system via pressing the L and U keys respectively";
  }

bool initModule()
  {
  initamPageModule();
  return 1;
  }

bool exitModule()
  {
  return 1;
  }
```

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    EXECUTIVE SUMMARY
//
// File Name: amPageModule.h
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

#ifndef _amPageModule
#define _amPageModule

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    FUNCTION PROTOTYPE SPECIFICATIONS
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

void initamPageModule();

#endif // _myKeyModule
```

```
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    EXECUTIVE SUMMARY
//
// File Name: amPageModule.c
//
// Author: CPT Stewart Liles, Naval Postgraduate School
//
// Description: defines methods pertinent to module loading and unloading
//
// September 1998, Masters Thesis
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    INCLUDES AND EXTERNS
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

#include "amPageModule.h"
#include "bbKeyboard.h"
#include "bbEventResponse.h"
#include "bbCallback.h"
#include "bbModule.h"
#include "bbMouse.h"
#include "npsWindow.h"
#include "npsViewport.h"

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
//    CODE
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

// ••••••••••••••••••••••••••••••••••••••••••••••••••••••
// Function Name: initamPageModule()
// Task:  initialize Module -- this is run only once
//    sets up keyboard callbacks
// Return Value: void
// ••••••••••••••••••••••••••••••••••••••••••••••••••••••

void initamPageModule(){
    void loadFunc(bbObject *object, bbData *data);
    void unloadFunc(bbObject *object, bbData *data);

    bbKeyboard        *keyboard;
    bbEventResponse   *eventResponse;
    bbCallback        *callback;

    keyboard = bbKeyboard::getInstance();

    // load module key
    eventResponse = new bbEventResponse(bbKeyboard::KEY_L |
bbKeyboard::CTRL_MASK | bbKeyboard::DOWN_TRANS);
    callback = new bbCallback();
    callback->setFunc(loadFunc);
    entResponse->addCallbackLast(callback);
    keyboard->addEventResponse(eventResponse);

    // unload module key
    eventResponse = new bbEventResponse(bbKeyboard::KEY_U |
bbKeyboard::CTRL_MASK | bbKeyboard::DOWN_TRANS);
```

70

```cpp
    callback = new bbCallback();
    callback->setFunc(unloadFunc);
    eventResponse->addCallbackLast(callback);
    keyboard->addEventResponse(eventResponse);

}//end initamPageModule


// *****************************************************************
// Function Name: loadFunc()
// Task:  callback function that loads modules
// Return Value: void
// *****************************************************************

void loadFunc(bbObject *object, bbData *data){
    int mouseInWindow();

    char  moduleName[64];
    char *ptr;

    if(mouseInWindow()){

        cout << "LOAD:Please enter the module's name : " << flush;
        cin >> moduleName;
        cout << endl;

        if (!strncmp(moduleName, "http://", 7)) {
            ptr = moduleName + 7;        // skip over http://
            ptr = strrchr(ptr, '/');  // make sure url well formed
            if (!ptr){
                cout << "  myDynamicPageModule: Bad command line parameter." <<
endl;
                cout << "  myDynamicPageModule: URLs must be formatted as
\"http://<host>[/path]/moduleName\" " << endl;
                cout << "  myDynamicPageModule: Example: bamboo
http://watsen.net/Bamboo/Modules/myRemoteModule" << endl;
                cout << endl;
            }
            *ptr = '\0';  // efficient hack
            if ( *(ptr+1) == '\0'){ // module can not be index.html!

                cout << "  myDynamicPageModule: Bad command line parameter." <<
endl;
                cout << "  myDynamicPageModule: URLs must be formatted as
\"http://<host>[/path]/moduleName\" " << endl;
                cout << "  myDynamicPageModule: Example: bamboo
http://watsen.net/Bamboo/Modules/myRemoteModule" << endl;
                cout << endl;
            }
            if (!bbModule::load(ptr+1, moduleName)){
                *ptr = '/';  // put it back
                cout << "  myDynamicPageModule: Unable to load " << moduleName <<
endl;
                cout << endl;
            }
        }
        else { // must already be on disk
```

```cpp
            if (!bbModule::load(moduleName, 0)){
                cout << "  myDynamicPageModule: Unable to load " << moduleName <<
endl;
                cout << endl;
            }
        }
    }//end if
}//end loadFunc


// *****************************************************************
// Function Name: loadFunc()
// Task:  callback function that unloads modules
// Return Value: void
// *****************************************************************

void unloadFunc(bbObject *object, bbData *data){
    int mouseInWindow();

    char      moduleName[64];
    bbModule *module;

    if(mouseInWindow()){
        cout << "UNLOAD:Please enter the module's name : " << flush;
        cin >> moduleName;
        cout << endl;

        module = bbModule::findObject(moduleName);
        if (!module){
            cout << "  myDynamicPageModule: specified module \"" << moduleName <<
" is not currently loaded...ignoring" << endl;
            return;
        }

        if (!bbModule::unload(module)){
            cout << "  myDynamicPageModule: Error unloading " << moduleName <<
endl;
            cout << "  Fatal Error - aborting executable!" << endl << endl;
            exit(0);
        }
    }//end if
}// end unloadfunc


// *****************************************************************
// Function Name: mouseInWindow()
// Task:  tells if mouse is in the correct window to receive keyboard
//    commands
// Return Value: int representing boolean
// *****************************************************************

int mouseInWindow() {
    int flag = 0;

    npsWindow             *window;
    npsViewport           *viewport;
```

```cpp
    window = npsWindow::findObject("AdminWindow");
    viewport = npsViewport::findObject("AdminViewport");
    float       x,y;
    x = bbMouse::getX();
    y = bbMouse::getY();
    bbScreen::normalizeVal(&x,&y);

    if(window){
        if (viewport->getWindow()->isValInside(x,y)){
            flag = 1;
        }
    }
    return flag;
}
```

APPENDIX C: GLOSSARY

- **attribute**
  - A named portion of an object state.
- **event**
  - A change of object attribute value, an interaction between objects, an instantiation of a new object, or a deletion of an existing object that is associated with a particular point on the federation time axis. Each event contains a time stamp indicating when it is said to occur (also see definition of message).
- **federate**
  - A member of a HLA Federation. All applications participating in a Federation are called Federates. In reality, this may include Federate Managers, data collectors, live entity surrogates simulations, or passive viewers.
- **federation**
  - A named set of interacting federates, a common federation object model, and supporting RTI, that are used as a whole to achieve some specific objective.
- **federation execution**
  - The federation execution represents the actual operation, over time, of a subset of the federates and the RTI initialization data taken from a particular federation. It is the step where the executable code is run to conduct the exercise and produce the data for the measures of effectiveness for the federation execution.
- **Federation Object Model (FOM)**
  - An identification of the essential classes of objects, object attributes, and object interactions that are supported by an HLA federation. In addition, optional classes of additional information may also be specified to achieve a more complete description of the federation structure and/or behavior.
- **interaction**
  - An explicit action taken by an object, that can optionally(within the bounds of the FOM) be directed toward other objects, including geographical areas, etc.
- **message**
  - A data unit transmitted between federates containing at most one event. Here, a message typically contains information concerning an event, and is used to notify another federate that the event has occurred. When containing such event information, the message's time stamp is defined as the time stamp of the event to which it corresponds. Here, a "message" corresponds to a single event, however the physical transport media may include several such messages in a single "physical message" that is transmitted through the network.

73

- model
  - A physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process. [DoD 5000.59]
- object
  - A fundamental element of a conceptual representation for a federate that reflects the "real world" at levels of abstraction and resolution appropriate for federate interoperability. For any given value of time, the state of an object is defined as the enumeration of all its attribute values.
- object model
  - A specification of the objects intrinsic to a given system, including a description of the object characteristics (attributes)and a description of the static and dynamic relationships that exist between objects.
- object model framework
  - The rules and terminology used to describe HLA object models.
- object ownership
  - Ownership of the ID attribute of an object, initially established by use of the Instantiate Object interface service. Encompasses the privilege of deleting the object using the Delete Object service. Can be transferred to another federate using the attribute ownership management services.
- Runtime Infrastructure (RTI)
  - The general purpose distributed operating system software which provides the common interface services during the runtime of an HLA federation.
- simulation
  - A method for implementing a model over time. Also, a technique for testing, analysis, or training in which real-world systems are used, or where real-world and conceptual systems are reproduced by a model.
- Simulation Object Model (SOM)
  - A specification of the intrinsic capabilities that an individual simulation offers to federations. The standard format in which SOMs are expressed provides a means for federation developers to quickly determine the suitability of simulation systems to assume specific roles within a federation.
- time management
  - A collection of mechanisms and services to control the advancement of time within each federate during an execution in a way that is consistent with federation requirements for message ordering and delivery.

LIST OF REFERENCES

[1]   Carlsson, C. and O. Hagsand (1993). <u>DIVE – A Platform fo Muti-User Virtual Environments,</u> Computing and Graphics Vol.17, No.6, pp.663-669.

[2]   Singh, G., L. Serra, W. Png and H. Ng (1994). <u>BrickNet: A Software Toolkit for Network-Based Virtual Worlds</u>, Presence, vol. 3, No. 1, winter 1994, pp. 19-34.

[3]   Macedonia, M., M. Zyda, D. Pratt, P. Barham, and S. Zeswitz (1994). NPSNET: <u>A Network Software Architecture for Large-Scale Virtual Environments</u>, Presence, Vol. 3, No. 4, Fall 1994, pp.256-287.

[4]   Watsen, K. and M. Zyda (1998). <u>Bamboo – A Portable system for Dynamically Extensible, Real-time, Networked, Virtual Environments</u>. 1998 IEEE Virtual Reality Annual International Symposium (VRAIS '98), Atlanta, Georgia.

[5]   Dahmann, J., <u>High Level Architecture for Simulation</u>, Defense Modeling and Simulation Office, http://hla.dmso.mil.

[6]   Adobe   (1997),   Photoshop   Software   Development   Kit, <u>ftp://ftp.adobe.com/pub/adobe/devrelation/sdk/photoshop</u>.

[7]   Netscape   (1997),   Navigator   4.0   Plug-in   Guide, <u>http://developer.netscape.com/library/documentation/communicator/</u> plugin/contents.htm

[8]   "High Level Architecture Object Model Template, V 1.0", Defense Modeling and Simulation Office, 15 August 1996, [http://hla.dmso.mil].

[9]   "High Level Architecture Rules, V 1.0", Defense Modeling and Simulation Office, 15 August 1996, http://hla.dmso.mil.

[10]   "High Level Architecture Interface Specification, V 1.0", Defense Modeling and Simulation Office, 15 August 1996, http://hla.dmso.mil.

BIBLIOGRAPHY

Adobe (1997), Photoshop Software Development Kit,
ftp://ftp.adobe.com/pub/adobe/devrelation/sdk/photoshop.

Carlsson, C. and O. Hagsand (1993). DIVE – A Platform fo Muti-User Virtual
Environments, Computing and Graphics Vol.17, No.6, pp.663-669.

Dahmann, J., High Level Architecture for Simulation, Defense Modeling and Simulation
Office, http://hla.dmso.mil.

"High Level Architecture Interface Specification, V 1.0", Defense Modeling and
Simulation Office, 15 August 1996, http://hla.dmso.mil.

"High Level Architecture Object Model Template, V 1.0", Defense Modeling and
Simulation Office, 15 August 1996, http://hla.dmso.mil.

"High Level Architecture Rules, V 1.0", Defense Modeling and Simulation Office, 15
August 1996, http://hla.dmso.mil.

"High Level Architecture Run-Time Infrastructure Programmer's Guide", Defense
Modeling and Simulation Office, 15 May 1997, http://hla.dmso.mil.

Macedonia, M., M. Zyda, D. Pratt, P. Barham, and S. Zeswitz (1994). NPSNET: A
Network Software Architecture for Large-Scale Virtual Environments, Presence, Vol. 3,
No. 4, Fall 1994, pp.256-287.

Netscape (1997), Navigator 4.0 Plug-in Guide,
http://developer.netscape.com/library/documentation/communicator/plugin/contents.htm

Singh, G., L. Serra, W. Png and H. Ng (1994). BrickNet: A Software Toolkit for
Network-Based Virtual Worlds, Presence, vol. 3, No. 1, winter 1994, pp. 19-34.

Watsen, K. and M. Zyda (1998). Bamboo – A Portable system for Dynamically
Extensible, Real-time, Networked, Virtual Environments. 1998 IEEE Virtual Reality
Annual International Symposium (VRAIS '98), Atlanta, Georgia.

Woo M., J. Neider and T. Davis (1997). OpenGL Programming Guide: The Official guide
to Learning OpenGL, version 1.1, Addison Wesely Developers Press, July 1997, ISBN 0-
201-46138-2.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
    8725 John J. Kingman Road, Ste 0944
    Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
    Naval Postgraduate School
    411 Dyer Rd.
    Monterey, California 93943-5101

3. Dr. Michael J. Zyda, Code CS/Zk 1
    Naval Postgraduate School
    Monterey, CA 93940-5000

4. Dr. Rudy Darken, Code CS/Dk 1
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93940-5000

5. Dr. Dan Boger, Code 32 CS 1
    Computer Science Department
    Naval Postgraduate School
    Monterey, CA 93940-5000

6. Capt. Jay Kistler, USN 1
    N6M
    2000 Navy Pentagon
    Room 4C445
    Washington, DC 20350-2000

7. George Phillips 1
    CNO, N6M1
    2000 Navy Pentagon
    Room 4C445
    Washington, DC 20350-2000

8. Mike Macedonia 1
    Chief Scientist and Technical Director
    US Army STRICOM
    12350 Research Parkway
    Orlando, FL 32826-3276

9. National Simulation Center (NSC)_____1
   ATTN:ATZL-NSC (Jerry Ham)
   410 Kearney Avenue --- Building 45
   Fort Leavenworth, KS 66027-1306

10. Director_____1
    Office of Science & Innovation
    OSI, MCCDC
    3300 Russell Road
    Quantico, VA 22134-5021

11. Capt. Dennis McBride, USN_____1
    Office of Naval Research (341)
    800 No. Quincy Street
    Arlington, VA 22217-5660

12. Col. Crash Konwin, USAF_____1
    DMSO
    1901 N. Beauregard St.
    Suite 504
    Alexandria, VA 22311

13. Sid Kissen_____1
    National Security Agency
    Attn: S312
    9800 Savage Road
    Fort George G. Meade, MD 20755